HETEROGENEOUS HIERARCHICAL MODELING FOR
KNOWLEDGE-BASED AUTONOMOUS SYSTEMS

By

VICTOR TODD MILLER

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE
UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1993

TABLE OF CONTENTS

Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

HETEROGENEOUS HIERARCHICAL MODELING FOR KNOWLEDGE-BASED AUTONOMOUS SYSTEMS

By

VICTOR TODD MILLER

August 1993

Chairman: Paul A. Fishwick
Major Department: Computer and Information Sciences

High autonomy systems generally require the use of multiple modeling formalisms and multiple levels of abstraction in order to describe their dynamic characteristics accurately and efficiently. It is often necessary to integrate several modeling formalisms if there is a need to reason about, simulate or analyze a system. Additionally, during development, the use of a hierarchical representation helps to organize the models more intelligently. Heterogeneous hierarchical modeling is a method which supports multiple representations and hierarchical development of knowledge-based autonomous system simulations.

In this context, hybrid model theory is developed as a theoretical representation which provides the necessary formality to meet the requirements of heterogeneous hierarchical modeling. Hybrid model theory is an alternative approach to combined discrete-continuous multimodel theories and subsumes most of the concepts in combined discrete-continuous system simulation. Hybrid model theory supports a new concept in heterogeneous hierarchical modeling called intramodel coordination. Intramodel coordination is a method in which the components of a model can be coordinated with other models. In this manner, hybrid model theory extends the notion of intermodel coordination in combined discrete-continuous system simulation. Intermodel

iv

coordination is a method in which two models can only interact through input and output. Furthermore, a hybrid model is a declarative representation of a system. By restricting the form of this representation, a hybrid model contains the data necessary information for a computer environment to perform symbolic, numerical and interpretative analysis automatically without additional effort from the user.

CHAPTER 1
INTRODUCTION

Heterogeneous Hierarchical Modeling

An essential part of any system description is the development of a model. Ideally, the model represents the behavior of the system under investigation at some level of abstraction. In the context of simulation, the process of developing this model is generally called simulation modeling or simulation methodology. The extent of the modeling process itself varies not only from individual to individual, but also from paradigm to paradigm. This work focuses on the issues of the modeling process in computer (digital) simulation. The domain is limited to systems for traditional engineering purposes. The processes and formalisms for biological, social and self-evolving systems may or may not be applicable to this discussion.

In this context, hybrid model theory is developed as a theoretical representation which provides the necessary formality to meet the requirements of heterogeneous hierarchical modeling. Hybrid model theory is an alternative approach to combined discrete-continuous multimodel theories and subsumes most of the concepts in combined discrete-continuous system simulation. Hybrid model theory supports a new concept in heterogeneous hierarchical modeling called intramodel coordination. Intramodel coordination is a method in which the components of a model can be coordinated with other models. In this manner, hybrid model theory extends the notion of intermodel coordination in combined discrete-continuous system simulation. Intermodel coordination is a method in which two models can only interact through input and output. Furthermore, a hybrid model is a declarative representation of a system. By restricting the form of this representation, a hybrid model contains the data necessary information for a computer environment to perform symbolic, numerical and interpretative analysis automatically without additional effort from the user.

The term modeling will be defined as a formalism and its associated methodology (if one exists). In simulation, most modeling techniques are low- to mid-level paradigms. At most, these techniques are stages in some potential methodology [Fis89b]. In the last decade, however, expanding the techniques of simulation methodology has received great attention [Cel82, Elz89, Pui89]. Additionally, new paradigms with associated methodologies have emerged and have been implemented [Nan87, Zei90].

One of the most import concepts to manifest itself from the research in simulation methodologies is the idea of a hierarchical model. Hierarchies have been used in highly abstract modeling environments [Cel92, Zei90], at the formalism level [Gor90] and at the numerical level [Syd82]. A hierarchical model is a model which has several different levels of representations and abstractions. These levels are related to each other by a hierarchy. The types of hierarchies most commonly used in simulation are structural, conceptual, and class hierarchies. A structural model decomposes the model into levels which resemble the actual system. Usually this requires that the system has well established physical attributes. A conceptual model may or may not have a decomposition similar to the system, but always has components which do not have physical counterparts in the system. These decompositions resemble the approach used in software engineering. A class model decomposition resembles the approach used in object-oriented programming. (The object-oriented class hierarchy actually originated from SIMULA 67, a language designed for discrete simulation [Bir79].) In a class hierarchy, the model is not decomposed hierarchically, but the objects used to described the model are created hierarchically.

A hierarchy is used to classify objects into types. It is important to distinguish the classifying of domain models as opposed to the classification of the formalisms used to describe domain models. A hierarchy which classifies vehicles (objects like truck, car, motorcycle, Buick, Honda, etc.) organizes the domain of vehicles. A hierarchy which classifies formalisms (state, functional, process, Petri net, control theory, semantic nets, expert system, etc.) organizes the domain of formalisms which are used to describe other domains. In this discussion, both hierarchies are used to organize concepts.

Another important concept which has emerged in simulation modeling is the use of multiple formalisms within one model [Pag89, Fis91b] . With the increased interest in hierarchical modeling, the need to represent efficiently each level of a knowledge-based autonomous system model (KAS model) has become important for many reasons. Some modeling formalisms capture certain aspects of system behavior better than others (developmental efficiency). Other modeling formalisms may provide the means to discern important features that are not evident in other formalisms (conceptual efficiency). The use and benefits of multiple models types has been investigated in theories such as multifaceted modeling [Zei84] and heterogeneous interlevel refinement [Fis91a].

The general concept of heterogeneous modeling draws upon research in multimodels, combined models, multifaceted models, homomorphic models, and abstract models. One of the ways of advancing the field of simulation requires improving the available modeling methods. Heterogeneous hierarchical modeling improves methods by aiding in the development, maintenance, simulation, and conceptualization of KAS models. It does this by providing a variety of succinct formalisms and the techniques for integrating them.

Submodeling is the process of either refining a model or specifying several alternative models for a model. Refining a model requires adding detail, for instance, refining a Petri net into a queuing model. Refinement of a model introduces subtleties of components of the model. Specifying alternatives for a model requires that only one alternative model be active at a time [Öre91]. Submodeling can require homomorphic behavior [Zei84, Sev91] between different levels of abstraction. Additionally, multiple formalisms include multiple types of simulation and integrating the taxonomy of models [Öre89b]. For example, Prähofer has integrated the representation of continuous and discrete event models in the same simulation environment [Prä91a, Prä91b].

Heterogeneous hierarchical modeling is a modeling process which supports multiple formalisms and supports hierarchical development as described above. Model evolution and engineering [Fis89a] are natural attributes of such a process. Heterogeneous hierarchical

modeling (HH modeling) allows an investigator to develop a model in a top-down fashion. The benefits of top-down design have been well established in many disciplines. The hierarchy not only aids the investigator in development and conceptualization of the model, but serves as a history of the development process itself. The history of a model's development may play an important role in the evolution or engineering of the model.

In any modeling environment, the investigator must have the maximum available flexibility in development while ensuring that inconsistencies within the model do not arise. Because checking for inconsistencies in large and multiple models can be tedious and complex, it seems natural to conclude that the development of methods for automating a HH modeling process will be useful. Two of the major inconsistencies in HH modeling methodologies are compatibility between levels of the model and compatibility among different types of models.

The number and type of formalisms in an HH modeling environment should provide the investigator with a sufficient range of choices (Petri nets, Markov systems, etc.). Of course, no modeling system will be perfect for every domain, but the interactions among a sufficient number of formalisms should give the investigator a flexible environment which can be applied to a very general domain.

Currently, many of the hierarchical systems allow only one type of formalism, for instance, hierarchical Petri nets or state machines. Many of the multimodel systems, such as GPSS [Sch91] or SIMAN [Peg90], allow only one level of abstraction. Combining hierarchies and heterogeneous models has just recently received attention in the literature [Fis91b]. A generalized and formal theory of HH modeling has yet to be developed. The motivation for the development of HH modeling is derived from several different sources. The contribution it can make to the modeling process in general has been speculated but not confirmed. This work is a direct result of these issues.

A growing but still underrepresented topic in modeling research is the ability to analyze a single model using symbolic, numerical, and interpretative techniques. Symbolic techniques are mathematical procedures which involve manipulating algebraic or differential equations. With the

increased use and availability of programs that do symbolic mathematics, it is becoming increasingly easier to automate symbolic analysis (especially of well-known modeling formalisms). Numerical techniques refer to traditional computational simulation methods and numerical approximation. Interpretation methods are those that are related to the field of artificial intelligence and knowledge engineering. These range from fuzzy or quantitative simulation [Fis91b] up to and including logic methods and semantic networks. In general, hybrid analysis (symbolic, numeric, and interpretative) requires different model specification strategies and processes for each type of analysis. This duplicates a substantial amount of effort on the part of the investigator. It also makes it impossible for information gained in one type of analysis to help or guide a technique in another type of analysis (unless the investigator transmits "by hand" this information from one modeling formalism to the other).

A legitimate approach to HH modeling is to develop a new modeling formalism which is oriented toward KAS models. In order to provide insights into the complex behavior of KAS models through simulation and reasoning methods, efficient and succinct representations must be used to describe all aspects of behavior which are pertinent to the investigator. It is unlikely that one modeling formalism would provide such a representation. This assertion is based upon the pragmatics of model building. Specific modeling formalisms are used by investigators because they are convenient to use, have preferable attributes, or fulfill some pragmatic requirement [Rot90]. There are clearly two dilemmas to investigators of KAS systems. First, since pragmatic issues are dictated by the investigator's preferences and convenience, and pragmatic issues vary within different sections of large complex systems, a single modeling formalism locks the investigator into a method which is neither preferable nor convenient.

The second dilemma concerns the trade-offs between convenience and generality. For example, queuing networks may be efficient for modeling arrival/departure behavior, but not general enough for the modeling of complex KAS models. Similarly, simulation languages such as GPSS are general enough to be used for almost all types of simulation, but lack the developmental efficiency, conceptual efficiency and convenience of mathematical-based

modeling formalisms. Additionally, simulation languages have traditionally lacked symbolic and interpretative analysis methods. In short, the more general a formalism becomes, the less efficient it is to use.

With this in mind, an HH modeling theory which is based on coordination of existing modeling formalisms is an attractive alternative to developing an all-encompassing, completely generalized, single formalism. Since modeling formalisms such as queuing networks and finite state machines have proven to be powerful methods, but limited to specific domains, a coordination of these modeling formalisms, which keeps the representational power of each formalism intact, should foster more complete investigations of complex high autonomy systems. Furthermore, by coordinating established modeling formalisms, the learning curve needed to understand the theory is reduced. HH modeling can be accomplished by letting the investigator use an appropriate modeling formalism to describe a particular component of the system and then allowing a coordination of this formalism with models that describe other system components. The investigator may also need to reimplement subcomponents of a particular model as new information is gained during development and analysis (perhaps with a different modeling formalism). Efficiency and succinctness are supplied by the mathematical formalism whereas the coordination of several formalisms increases the generality.

Motivation for Research in HH modeling

There are several research issues which motivate the development of heterogeneous hierarchical modeling. Among these issues, the most basic, yet very important, is advancing the field of simulation. Ören [Öre89a, pg. 30] writes,

> Since use of models is essential in simulation, advancements can be achieved
> by exploring advances in model-based concepts such as: modeling formalisms;
> modeling environments; model-based management; and symbolic processing
> of models.

By using multiple models, HH modeling expands the notion of a modeling formalism. Research in multiple models is not new. However, it is still in the early stages of development and requires further exploration. Combining multiple models within a complex hierarchical structure adds new problems and complicates current problems of multiple model research.

The hierarchical modeling process is related to research in model-based management and modeling environments as described by Ören. The hierarchy serves to manage the development and structure of the model. The top-down approach commonly used with hierarchical models helps to prescribe management methods. Furthermore, a hierarchy describes the traversal path a modeling environment might use. Any environment which aids the investigator by allowing perusal of the model structure can use the natural structure of the hierarchy to guide the investigator.

One of the most important motivations for research in HH modeling is the contribution it makes to knowledge-based simulation [Fis91a] and qualitative simulation [Fis91b, Kui89]. In knowledge-based simulation, information about the model is used to aid the investigator by suggesting alternative formalisms, answer questions about the model, increase semantic relationships between parts of a model, and help develop intelligent agent and goal-directed systems.

HH modeling establishes formal relationships and clarifies the semantic relationships between different levels of abstractions in a model and different types of formalisms. For instance, a continuous model which has well-defined system states can be described by a state machine at the top level of a hierarchy and difference equations at the bottom level [Fis91b]. The formal relationships of HH modeling ensure correctness of the model. The hierarchy clusters information which is useful in grouping behavior, and the type of model specifies semantics about the particular level. A knowledge-based simulation and environment requires such information in order to suggest improvements, guide semantic development, or analyze autonomous decision-making models. Therefore, a strong motivation for developing HH modeling is to help establish a foundation for knowledge-based simulation.

An important part of research in qualitative simulation is the ability to ask questions that require varying degrees of specification. An investigator may be interested in symbolic information in one question (will the ball bounce and if so, will it bounce high) or numerical information in other questions (how many times will it bounce and how long will it take to stop). These questions require different levels of abstraction and different levels of implementation. A hierarchical model provides, by its very nature, different levels of abstraction with corresponding formalisms capable of providing data appropriate for that level of abstraction. Similar to the motivation for knowledge-based simulation, HH modeling development helps to establish a foundation for qualitative simulation.

Contribution

The motivations in the previous section identify two general contributions. First, by developing formal and semantic relationships between different types of formalisms, HH modeling contributes to research in expert systems and knowledge-based simulation. Second, by developing the formal relationships between abstract levels in hierarchical models, HH modeling contributes to research in qualitative simulation. Although these two contributions are important, they are not direct contributions to the state-of-the-art simulation environment. HH modeling will form a solid foundation for knowledge-based simulation systems.

The direct contributions HH modeling can offer to state-of-the-art simulation environments are as follows:

- Most simulation environments and libraries offer an investigator multiple models from which to choose. However, they do not help the investigator use them. How different models in the same simulation may interact is left up to the investigator. No facilities to check for inconsistencies among interacting formalisms are provided.
- Designing simulation models is currently performed in an ad-hoc manner. HH modeling provides a structured way to create models efficiently in a top-down fashion.

- Changing simulation models as new data are collected or new constraints are added can significantly alter flat models. The hierarchy of HH modeling serves to isolate independent parts of a model and therefore make them easier to maintain.

- The hierarchy of formal models allows for incomplete models to be simulated. This gives the investigator feedback early in the simulation development phase.

- HH modeling provides a mechanism to ask questions not only about the results of the simulation, but about the model being simulated. Additionally, the level of complexity of the question dictates the level of abstraction used in the simulation, and the level of ambiguity of the results of a simulation dictates the level of abstraction used to answer the question.

- A generalized theory will make additional formalisms easier to include.

- HH modeling allows formalisms to interact with each other instead of combining different formalisms into one. The complexity of modeling a large system is therefore broken up not only hierarchically but also conceptually.

Related Work and Topics

There are three fields of study which are indirectly related to the work presented in this research. These fields are artificial intelligence, software engineering, and knowledge-based simulation. The exact relationship is probably a matter of philosophical debate. However, the cross-fertilization of ideas between these fields is prevalent through out the literature [Fis92].

Within AI, the work in qualitative reasoning [Bob86] is directly related to HH modeling. Qualitative reasoning, in general, is any type of formalism which is an abstraction of algebraic or difference equations. The recent work of Forbus and Falkenhainer [For90] and Kuipers [Kui89] is a good representative of the relationship between HH modeling and qualitative reasoning. In Forbus's paper, self-explanatory simulations based on qualitative process (QP)

theory [For86] are used to answer a variety of questions about a model at several different levels of abstraction. States in QP theory are based on equation limits. In Kuipers's paper, a qualitative graph of data flow with constraints is used to reason about incomplete systems. Qualitative simulation (QS) [Kui86] is used as the reasoning formalism. States in QS are based on specified landmark values.

Knowledge representation (KR) is loosely related to this work. Typically, KR focuses on static relationships between objects. Frames [Hay79], semantic nets [Fin79], inheritance [Eth83] and logic [Moo82] all play roles in reasoning about static properties. Although temporal reasoning research is done in AI, for example Allen's work [All83, All84], dynamic properties about system state is confined to qualitative reasoning. Reasoning and questioning are considered synonymous in this work. Therefore, when a question is posed to obtain properties of a model, reasoning is taking place.

In software engineering (SE) the use of formalisms to describe operational behavior is common and is found in literature explaining SE fundamentals [Ghe91]. Object-oriented programming is also an important modeling topic in SE [Har89]. This emanates from software maintainability and reuse issues in which object hierarchies tend to assist. A good representative of the related work in SE is the new book by Rumbaugh et al. [Rum91]. Rumbaugh's methodology uses objected-oriented concepts to define static properties which are very much like semantic nets in AI. The dynamic properties of a software project are described using state and process modeling techniques. Although this methodology is not formal (theoretically or computationally), it is a thorough embodiment of process modeling.

Within simulation research, any topic which uses object-oriented concepts or AI concepts is related to this work. Expert systems to analyze output, suggest models, or analyze sensitivity need to be able to ask questions (reason) about models and be assured that the answers have a sound foundation. Integrated software environments which help an investigator construct models must be able to ask questions (reason) about models and be assured that the answers have a sound foundation.

All these areas of research have at least one thing in common: they must refer to abstract properties about complex models. Yet, the formal theory of abstract, hierarchical, or heterogeneous models is relatively unexplored. Sevinc [Sev91, pg. 1118] recognizes this when referring to model abstraction. He states,

> No complete theories of model abstraction exist, nor does any sufficiently general procedure. The field, with only less than a half a dozen published articles, is wide open.

There is a distinction here which must be made. Sevinc refers to the simplification of a preexisting model. HH modeling is just the opposite, the refinement of a more detailed model from an abstract model. However, the theory used to describe homomorphic behavior as described by Sevinc should be independent of the development methodology.

Directly relating to this work is research by Zeigler [Zei76], Wymore [Wym86], and Sevinc [Sev91]. These works consider homomorphisms, or derivations thereof, as the basis for model abstraction. In particular Zeigler's work in discrete event system (DEVS) [Zei84][Zei90] and Prähofer's continuous-discrete system [Prä91a] were considered as a starting point for this work. Three important differences should be pointed out. First, HH modeling extends this research to nondeterministic formalisms. Second, hierarchies of model formalisms dominated this work. Zeigler's and Prähofer's hierarchies center around the specific domain in question. These hierarchies are formed by coupling models together. Without the lower level models, no simulation can be preformed. In HH modeling, the goal is to simulate incomplete abstract models which generalize some, for the present, unknown behavior. Third, HH modeling allows a top-down and bottom-up approach to developing models whereas DEVS is a bottom-up approach.

Almost without saying, this work is a spin-off of Fishwick's research, so much so that it would be impossible to enumerate. However, the relationship between SE, AI, and simulation [Fis89b] and heterogeneous models [Fis92] are of particular importance.

Outline

In Chapter 2, the underlying modeling paradigm is introduced. A basic introduction to modeling, simulation, and system theory also is given. This provides a review of the basic concepts that are used in later chapters. Also, notation and meanings of important terms are defined. General system theory (GST) will be the foundation upon which a theory of HH modeling is developed. However, the GST definition does not have sufficient structure for what is called component coordination (GST does handle model coordination and was therefore a good starting point). Also, GST does not clearly integrate with domain independent knowledge-based reasoning techniques. Therefore, in order to support HH modeling, GST has been extended by including connectivity and abstraction concepts.

In Chapter 3, the formalisms chosen to represent modeling types, their use in simulation, their basic theoretical foundations, and their system description are presented. Specifically, five formalisms will be discussed: automata theory, Markov systems, Petri nets, queuing networks, and control theory. With these five formalisms, a sufficient spectrum of formalism types will be available to model a complex environment. However, these types are conceptually different enough to provide nontrivial problems in their integration within a unified framework.

Chapter 4 presents the modeling paradigm developed to support HH modeling. The modeling paradigm is called hybrid model theory. This presentation includes a formal characterization. Hybrid model theory is a direct attempt to simultaneously embrace two themes which are directly related to HH modeling. First, it expands, clarifies, and establishes a solid mathematical foundation for the notion of heterogeneous refinement as introduced in Fishwick and Zeigler [Fis92]. Second, hybrid model theory furnishes a premise for hybrid analysis of a system represented by a heterogeneous refinement model. Some minor modifications of the formalisms presented in Chapter 2 are made. This allows for the coordination between formalisms. The modifications will not alter the behavior of the formalisms. In some cases, however, the computational power of the formalisms may increase.

In Chapter 5, the benefits that HH modeling using hybrid model theory provide are demonstrated by the modeling of an automated flexible manufacturing system (AFMS). Traditional formalisms such as Petri nets, Markov systems, and block diagrams are used to create a heterogeneous hierarchical model efficiently. The symbolic and interpretative analysis methods are emphasized in this chapter. The numerical analysis is discussed briefly.

A conclusion and a summary are presented in Chapter 6. This includes the problems and shortcomings of hybrid model theory. Additionally, a brief discussion on how hybrid model theory provides a foundation for hybrid analysis is presented.

CHAPTER 2
BASIC CONCEPTS

Modeling and Simulation Concepts

There are different definitions for many of the general terms used in modeling and simulation. The definitions presented in this section may or may not be similar to common interpretations (although most are very similar). It is particularly crucial that the scope of the definitions presented be understood. For instance, a description of a model typically implies that the model has a time base [Wym77, Zei76]. The interpretation of a time base can vary between formalisms (e.g., between Petri nets and control theory). When using different types of models together, the meaning and scope of time must be expanded to accommodate an appropriate range of usages.

The most basic and most difficult term to define is model. For the purposes of discussion, a model is a representation or reproduction of a concept or physical object. The representation must have a formal description, that is, well-defined terms, entities, and operations on those entities. It may appear that representations which are only well defined unjustly restrict the numbers and types of models. However, as will be demonstrated later, a heterogeneous hierarchical model must maintain a mapping between levels of abstraction. Therefore, a representation must be well defined. Consequently, a well-defined representation is also called a formalism.

Mathematically, such a lose definition of a model is unusable. Therefore, a recursive definition of a model based on a system will be given. However, only a conceptual description is given here. This definition will be refined (i.e., well defined) in Chapter 4. Given a finite set of primitive (atomic) systems, a model is defined as

14

1. An atomic system

2. A structured collection of atomic systems

3. A structured collection of models.

The specifications of "structured collection" are given in Chapter 4. The above definition differs from traditional model definitions in systems theory. Here, all models must be built upon a finite set of atomic models (a finite set of primitives). Note this is different from a finite set of model types or classes.

As stated in the introduction, modeling will be defined as a formalism and its associated methodology. However, no commitment to either prescribed methodology (e.g., top-down or bottom-up) is adopted in the theory presented in Chapter 4. The emphasis is on a top-down methodology, but this is for exemplification purposes only. The process of development is undefined in terms of how is it performed. What is to be done in the development process is quite clear; a model is to be created. As previously implied, the ability to model hierarchically (top-down or bottom-up) provides a mechanism for a methodology, but does not force a methodology to use that mechanism in any prescribed way.

The term simulation means not only the prevalent processes such as queuing networks, Markov chains, etc., but also processes such as expert systems. Conceptually, an expert system is a simulation of a thought process. Expert systems have well-defined entities and a well-defined operation (rules and an inference engine). Rothenberg discusses this issue in greater depth and in a more general sense in "Artificial Intelligence and Simulation" [Rot90]. Knowledge-based simulation (KB Simulation) will be used to refer to the simultaneous activities of analysis, simulation, and interpretation of dynamic models. This definition is generally consistent with current literature on KB simulation [Fis91a].

A diagram of the relationship among some of the terms introduced is depicted in Figure 2.1. This type of configuration is thoroughly discussed in Zeigler's book [Zei76]. A system is represented by a model. The model is constructed through a modeling methodology and

processed by either a human or a computer. The symbolic analysis of models can be aided by the computer. Likewise, techniques in AI have begun to automate the qualitative interpretation of models [Fis91b]. The problem with simultaneous analysis, simulation, and interpretation arises in the formal representation of different models. Heterogeneous hierarchical modeling is a theory in which all three types of activity are supported. Figure 2.2 shows the representation of these combined activities.
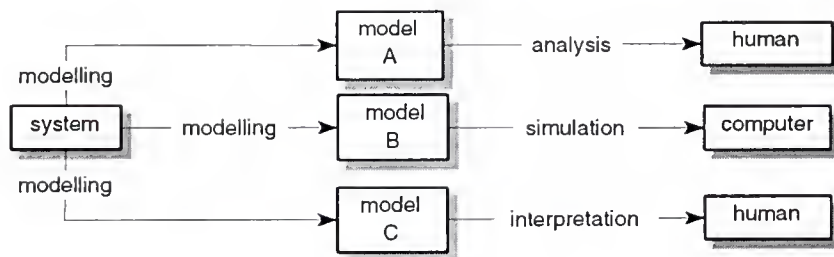
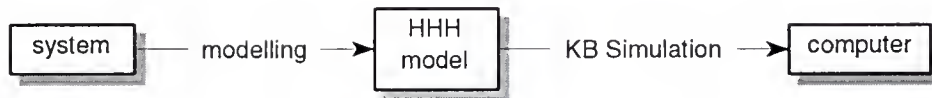

Figure 2.1 Traditional Modeling

Figure 2.2 Hybrid Modeling

Two important aspects of modeling which are not explicit issues of this work are validation and verification. Validating a model with a system ensures that the model represents the system's behavior to an adequate degree of accuracy. Verifying the model with the computer/human ensures that the formalisms are processed correctly. Although it is quite probable that the development of HH modeling will impact validation and verification, they are not discussed explicitly.

The relationship between system and model is further refined by establishing a conceptual view of a system (Figure 2.3). A system exists in an environment. The boundary between a system and an environment determines what is to be modeled. In simulation modeling, the environment has no representation and the system is represented by the model. The interaction between a system and the environment at the boundary is represented in a simulation modeling by input to or output from the model.
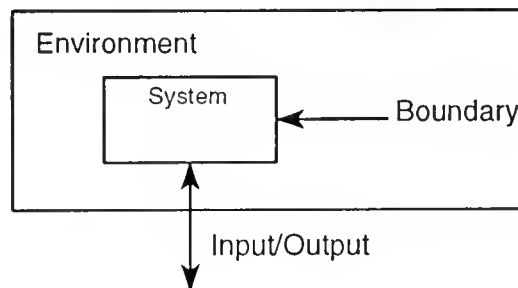

Figure 2.3 Conceptual View of a System

A system may be considered to be in one of several states at any given time. Typically, this is conceived of as a set of variables with each unique set of assignments to those variables being a state. A static system does not change with time while a dynamic system changes with time. Alternatively, a static system can be viewed as a dynamic system at a particular point in time. If a system has a unique set of outputs for each set of inputs, then the system is said to be deterministic. If the output of a system cannot be precisely predicted (or is random), then the system is said to be stochastic. Simulation generally deals with dynamic systems. However, KB simulations may include static systems.

An event occurs in dynamic systems when the system changes state. If the system is changing state continuously over time, then it is a continuous system. Systems which change only at specific points in time are discrete-event systems. Similar definitions can be found in Bank's book on discrete-event simulation [Ban84].

Formalism Classifications

There are two general classifications of formalisms that will be modeled in Chapter 4: state (operational, declarative) and functional (process, procedural). Together these general classifications cover a large number of specific formalisms. A state formalism represents states as entities. A simulation progresses as the model moves from one state to another regardless of whether time is elapsing. Figure 2.4 shows an example of a state model.
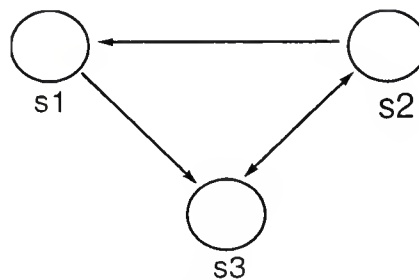

Figure 2.4 Typical State Model Diagram

The three entities (s1-3) in Figure 2.4 represent system states. An arrow represents moving from one state to another . Each state is exclusive of each other (although some formalisms may allow parallelism). An example of a state model might be the states of a drilling machine: working, turned-off, or under-repair.
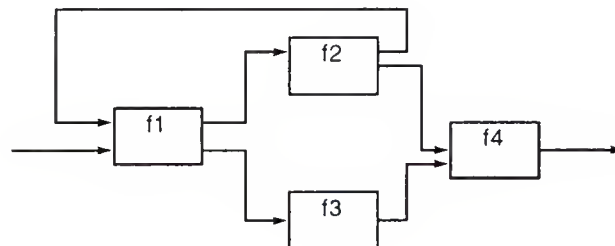

Figure 2.5 Typical Functional Model Diagram

A functional formalism is depicted in Figure 2.5. This is very similar to a data flow diagram. Each block represents a mapping (f1-4 in Figure 2.5) which transforms input to output. The state of the system is represented by the collection of internal states of each block. The arrows represent data transfers from one block to another.

An example of a functional system would be an electrical circuit. Each block represents a component: resister, capacitor, etc. The data transferred is the current (electrons). It should be noted that functional systems are commonly used as parallel models; this is complementary to state systems which are commonly sequential.

Hybrid Model Theory

Hybrid model theory is a direct attempt to simultaneously embrace two themes which are directly related to HH modeling. First, it expands, clarifies and establishes a solid mathematical foundation for the notion of heterogeneous refinement as introduced in Fishwick and Zeigler [Fis92]. In Fishwick and Zeigler's presentation, the concept of heterogeneous refinement was described as a method which helps bridge the gap between AI and simulation models in a formal manner. However, the refinement process was carried out "by hand." Hybrid model theory expands the concept and provides a foundation that allows heterogeneous refinement to be automated. Second, and most important, hybrid model theory furnishes a premise for hybrid analysis of a system represented by a refined multimodel. The extent of hybrid model theory encompasses a single model. This is an augmentation to theories, such as general system theory, which deal with classes of models.

It should be noted that hybrid model theory is a foundation which allows HH modeling to be implemented. There are certainly other approaches. However, hybrid model theory is an approach much like compiler theory. All programming languages can be described by compiler theory, yet there are many different types of programming languages which suit different purposes. The intention with hybrid model theory is similar; it is not expected that investigators

will use hybrid model theory as a formalism. Hybrid model theory is used to explain, mathematically, the commonalities and differences between modeling formalisms. With this foundation, the coordination of different formalisms such as Petri nets and block diagrams can be substantiated since the relationship between them has been formally established.

General System Theory

There are several different ways each formalism can be represented (presented in the next chapter). This produces a large permutation of coordination techniques. General system theory will be used as a starting point towards developing a common representation for the formalisms which are presented in this work. This section only introduces the mathematical foundation of systems theory for background and informational purposes. Most of the material described here is derived from Wymore's book [Wym77].

A system is a 6-tuple $Z = (T, I, S, A, B, \delta)$, where

T is the time base,

I is a nonempty set called the input,

S is a nonempty set called the system states,

A is an admissible set of input functions $f: T \rightarrow I$,

B is a set of functions $f: S \rightarrow S$ called the Behavior, and

δ is a function $f: A \times T \rightarrow B$ called the transition function.

The time base, T, is typically the reals ($\mathfrak{R}$) or the integers ($\mathfrak{S}$). When $T = \mathfrak{R}$, the system is said to be a continuous system. When $T = \mathfrak{S}$, the system is said to be a discrete system. The system can be considered to be like a function invocation, when the input set A along with a time t is given. The set of system states (S) varies greatly from formalism to formalism; however, it usually has the structure of an n-tuple or vector. For instance, the states of a state machine are usually represented by a 1-tuple (simple set), and the states of a continuous system are typically represented by a vector space on a field (such as $\mathfrak{R}^n$).

The admissible input functions represent the class of input schedules or input histories. Given a time segment t, a function in A gives the input presented to the system. This implies that the inputs to the system must be predetermined in order to analyze the system. The behavior functions (B) define the class of system sequences (discrete systems) or trajectories (continuous systems). The transition function generates a behavior function for a given input function and a time segment. Given an input function, initial conditions, and a transition function, the behavior of the system is completely deterministic.

The relationships between systems theory and the simulation concepts described in the last section are fairly clear. A simulation model is a super-set of a system. Both a model and a system have inputs, states, and behavior. One can extend a system structure to include output by adding the following definitions

O   is a nonempty set called the output and
λ is a function f: S x T $\rightarrow$ O called the output function.

A major distinction between a simulation model and a (classical) system is that a simulation model can be nondeterministic. However, in an abstract sense, there is a close correspondence between model and system. For example, in Figure 2.4, the state model consists of states with arcs labeling transitions from state to state. In systems theory, the model in Figure 2.4 is

S = {s1, s2, s3} and
δ (A, t)(s1) = s3; δ (A, t)(s2) = s1; δ (A, t)(s2) = s3; δ (A, t)(s3) = s2.

Similarly, in Figure 2.5, a system can be derived by letting S be the cross product of the functions (Π F_i) and δ (A, T) be the set of equations. Fishwick [Fis91b] presents a similar description of simulation modeling with systems theory.

Basic Modeling Paradigm

In the next chapter, the basic mathematical foundation of the formalisms used in this work is presented. Although one can find theoretical extensions of these formalisms in the modeling literature, this is nothing more than an attempt to combine formalisms (multimodels). This research uses an alternative approach. Instead of forcing a formalism to include other theoretical and semantic aspects, each formalism remains as close to its simplest or most common form, and a method is developed in which these different simple formalisms can be used together.

The foundation for this approach is based on three premises. First, the formalisms in their simplest state are well understood: Why develop or extend (yet another) formalism that is not well known when two well-known formalisms already exist? Second, researchers and investigators already use these formalisms. Not only are the formalisms understood but they are used frequently. Third, and most important, combining different aspects of different formalisms increases complexity at one level of conceptualization. By keeping each formalism separate, and introducing a simple way to interconnect them, the complexity has been separated into distinct parts. Consequently, inefficiencies in implementation may exist. However, efficiency in modeling can be improved. An underlying assumption here is that human time is more valuable than computer time. The compilation of a model (implementation) can be carried out by the computer whereas (at the moment) modeling is done by humans.

The distinction between a formalism and a theory is defined as a difference in generality. A formalism (Petri net, state machine) has relatively clear semantics pertaining to its use and dynamic properties. A theory (system theory, computation theory) is a more generalized mathematical system which usually can describe any known formalism. Because of the generality, automated analysis is typically infeasible.

Five common modeling formalisms are used as representatives of different modeling techniques. These fall into the two general classes above: state models or functional models. State machines and Markov systems were chosen as examples of state model formalisms. Queuing networks, Petri nets and block diagrams (control theory) were chosen to represent functional

model formalisms. The diagrammatic aspects of each specific modeling approach is preserved.
There is no attempt to homogenize modeling or to force all models to look like either data flow
diagrams or state transition networks. All model types have an equivalent graph or network
representation. This is necessary in order to support knowledge-based reasoning methods
(interpretation). However, this has not reduced the effectiveness of the theory presented since
many modeling formalisms have graph or network equivalents.

More specifically, the proposed paradigm requires that a formalism be represented by a
directed graph. Arcs (edges) which lead out of a node are output arcs and arcs leading into a node
are input arcs. Nodes in the graph represent either computational or storage models. As with
most theories of modeling, there are two basic types of models: atomic and structured. However,
in hybrid model theory a state machine, Petri net, etc., are not atomic models but structured
models. In hybrid model theory, structured models are made up of at least two hierarchical levels.
The first level is called a controller model. As will be shown, for a variety of formalisms only
three controller models are necessary. The second level in the hierarchy is made up of atomic
models called component models. This split-level approach to models is demonstrated in Figure
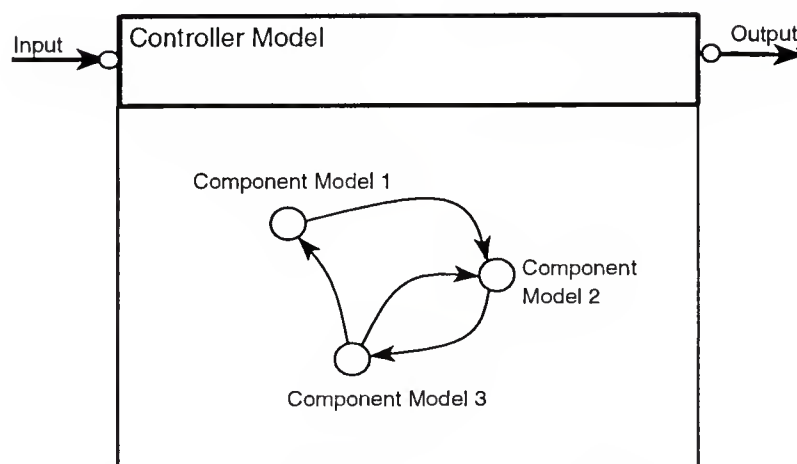2.6.

Figure 2.6 Two Level Representation of a Model Formalism.

As can be seen from Figure 2.6, data (or control) input and output are directed into the controller model. The component models (nodes in the graph) may or may not have data input and output. Depending on the type of controller, edges in the graph will either indicate control flow or data flow. This dual functionality has been captured in the controller model's interpretation of its components. Only under direction of the controller model is data input and output passed down to and up from the component models. This, at first, gives the impression of being very inefficient. However, when the model is compiled for numerical analysis(simulation), this inefficiency can be removed if and only if there has been no submodeling. When using interpretation techniques, this split-level method allows for more generalized knowledge. When analyzing the model symbolically, it allows combined results from different types of symbolic analysis. Chapter 4 introduces hybrid model theory in terms of numerical analysis (simulation). Chapter 5 shows by example how symbolic and interpretation analysis can be performed.

Formalisms are classified based upon three attributes: 1) how they use time, 2) the type of data they use, and 3) the type of controller. Hybrid model theory supports four types of controllers: parallel, state, selective, and group. All four of these controllers contain the connectivity of the components they control (the graph). A parallel-controller model controls component models in which all components are active simultaneously. Edges between the components are interpreted by the controller as data paths. Formalisms which have this type of controller are block diagrams, confluence graphs, bond graphs, and neural networks. A state-controller model controls components in which only one component can be active. The controller, under direction of the components, keeps track of the current active component. Edges between the components are interpreted by the controller as control paths. Formalisms which have state controllers are Markov systems and state machines. The selective controller is the most complex. This controller controls two types of component models: functions and storage. A selective controller first determines which function components can be activated and then nondeterministically chooses one of them to activate. The function components may use any of the storage components for data input and output. Edges between the components are interpreted

by the controller as data flow. Formalisms which use selective controllers are Petri nets, queuing networks, and expert systems. A group controller is a parallel controller in which the components are structured models. The group controller allows hybrid model theory to encompass traditional model coordination (coupling) and will only be briefly discussed.

The type of data which may be used by a formalism has two general attributes: value and time. Each of these attributes may be either continuous or discrete. This expands the typical continuous versus discrete concept of a signal in system theory. A discrete signal is too ambiguous of a categorization when combining symbolic analytical techniques and for interpretation techniques of different formalisms. It must be known whether a signal is discrete (continuous) over its values and over time.

The third element which classifies a formalism is the way in which time is used. There have already been significant advances in combined discrete-event and continuous model simulation through the use of time bases [Prä91a]. This forms the foundation for hybrid model theory. However, the time description is extended to include elements necessary for symbolic and interpretation methods. This extension is called a time domain. The concept of a time base which is used in system theory becomes one of five elements used in a time domain, the most important of which is the time map function. The time map of a time domain is a function from the reals into the time base of the model. This allows coordination of all models with a common time base. Each model is responsible for mapping the common clock into local time. This concept, along with local model states, allows hybrid model theory to be easily translated into a distributed simulation when numerical analysis is required. Thus, there is no main event queue during numerical analysis (simulation). All events are stored locally in a model and coordinated by a common clock.

The other elements of a time domain relate information concerning the semantics of the time base. Currently, there are three elements: a zero point, a delta time, and a magnitude function. The zero point signifies the minimum time required for a model to change an output signal given

a change in the internal state. The delta time signifies the minimum time required for a model to change its internal state. The magnitude function maps a time from the time base into the integers. This function permits a model to specify significant magnitude changes in time.
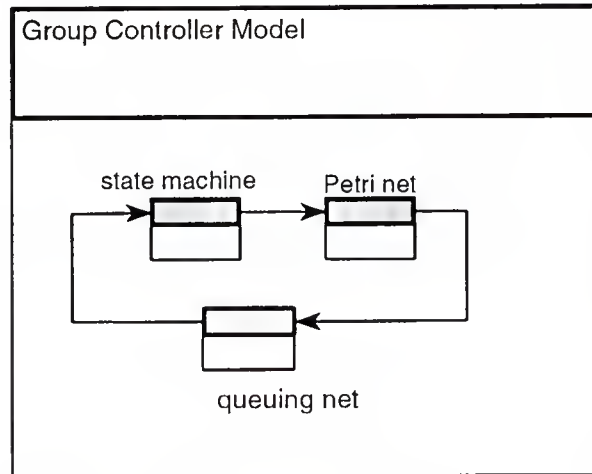

Figure 2.7 Intermodel Coordination

Intermodel coordination is another term for model coupling [Zei84, Wym77]. Because hybrid model theory has incorporated system theory, this type of coordination will not be extensively explained. Model coupling can be found in most system theory literature. From Figure 2.7, it should be clear that complex models of varying types can be coordinated through their data input and output. A collection of these models can then be grouped into a new model. An important advantage in hybrid model theory is that the intermodel coordination need not be static. That is, during execution or analysis, since the controller model contains the connectivity along with the functionality, the couplings can be dynamic; the controller can manipulate the connectivity of its components. This may become useful in certain types of neural network formalisms, for instance, as weights between neurons may become zero.

Intramodel coordination involves the replacement of a component model with a new structured model. The model hierarchy is therefore a structural or conceptual hierarchy. This is

quite different from intermodel coordination where, for instance, the output of a state machine is the input to a block diagram. In intramodel coordination, for example, a state component in a state machine controller model is replaced by a block diagram model [Fis92]. The controller model of a state machine essentially keeps track of several component models. Whether these components are simple state models which are based on conditions (input = 'a', etc.) or complex models such as block diagrams is inconsequential to the controller. The only requirement is that the communication between the controller and its components be standardized in a formal protocol. The same type of argument holds true for parallel, group and selective controllers. In Chapter 4, the theoretical details of controller-component intra model coordination are presented.

Definitions

Before continuing into more formal concepts, a few basic definitions and their designations are introduced. The meanings are generally well known, but it is essential that the interpretation of the designations used be clear. Therefore, they are presented here instead of being put into a key of symbols.

The unique symbols used are self, true, false, $\emptyset$, and $\dagger$. The empty set $\emptyset$ and the booleans true and false have their usual meanings. The symbol self is used to designate a symbolic reference to an entity which uses it. The symbol self is further explained in Chapter 4 where it has a special meaning in hybrid model theory. The symbol $\dagger$ is used to represent the notion of undefined (general math), null (programming), empty-string (automata), transient state (circuits), and bottom (programming theory). Depending on the domain, the appropriate term is used.

As a standard throughout this work the following notation is used. The only notable difference is that all function invocations are designated by square brackets [].

    [ ]   function application f [i]
    { }   general set
    ()   ordered set - sequence

< >   structured set and named set

Π_i cross product over i sets

Additionally, the following special sets are defined.

Z   Integers $\cup$ {†}

N   Z+ nonnegative Integers $\cup$ {†}

R   Reals $\cup$ {†}

B   Booleans {true, false} $\cup$ {†}

M   Model $\cup$ {†}

Two points are important here. First, all these sets are unioned with the undefined symbol {†}. Because these sets are usually used to specify values of variables, any variable in any model can be assigned the undefined value †.

Time Domains

The standard system theory notion of a time base will be used to specify the range of time values used by models in hybrid model theory. A time base is a structure consisting of a set and two operators: addition and comparison. The addition operator and the set must be an abelian group. The comparison operator and the set must form a linear order which is preserved under the addition operator.

Time Base $<T,+,\leq>$

$<T,+>$ an abelian group

$<T,\leq>$ a linear order preserved under +

Typical time bases are the reals $\mathfrak{R}$ and integers $\mathfrak{Z}$ with + and $\leq$ defined appropriately. These time bases are abbreviated by $T_{\mathfrak{R}}$ and $T_{\mathfrak{Z}}$. When a model or formalism has no time base, the null time base can be used. It is defined as $T_{\dagger}$.

Null Time Base

$T = \{\dagger\}$

Group $\dagger$ = identity, operator $\dagger + \dagger = \dagger$, inverse $\dagger = \dagger^{-1}$

Linear Order $\dagger \leq \dagger$

A time domain is built upon a time base. During the interpretation of a model, information about how the time is used by the model must be present. The time domain will serve this purpose. A time domain is a named set (a special kind of structure introduced shortly) which consists of five elements: time base, delta time, zero time, time map, and a magnitude function. A time domain TD is structured set <T, dt, zero, t, m> such that

T   Time base

dt   small time in T, i.e., a significant change in time

zero $t \in T$ such that everything < t is considered zero

t[]   time mapping $\Re$ -> T

m[] magnitude function T -> $\mathfrak{S}$ such that m[zero] = 0

The use of a time domain can be exemplified by the following two examples. Although the time base is the same, there are significant differences in how time affects human and computer systems.

Human Time	Computer Time
$T = T\Re$	$T = T\Re$
dt = 10 millisecond	dt = 1 picosecond
zero = 100 milliseconds	zero = 1 nanosecond

t[] = identity   t[] = identity

m[r] = integer[r/10*zero]   m[r] = integer [r/2*zero]

The zero time stipulates what times are to be considered as instantaneous. That is, in times less than zero the system cannot react to input. Note that this is different from the delta time dt. The delta time indicates what times are significant in changes in state. For instance, it is assumed that a human can sense things in 10 milliseconds but cannot react until 100 milliseconds. Likewise, in a picosecond, changes in transistors are important, but a cpu reacts only in nanoseconds (i.e., memory accesses).

The time mapping is used to relate all time domains to $T_\Re$. This will be further discussed in Chapter 4. The magnitude function is used to signify a constant state between systems. For example, for a time period of 0.9 seconds, the human magnitude function m[0.9] = 0 while the computer magnitude function $m[0.9] = 450x10^6$. For all practical purposes, in a time period of 0.9 seconds, a computer system can assume a human system is constant. The magnitude comparison can be used to circumscribe the system when any of the three types of analysis (symbolic, numerical, interpretation) are required.

Named Sets

In system theory, a convenient representation of assignment is represented by a structured set [Zei76, Zei84]. In this work, these sets are referred to as named sets. Formally, a named set is a structure <S, V, R, A> with

S - a set (entities)

V - ordered set (parameters)

R - indexed set (V is the index) R is the range

A - assignment A: S -> $\Pi_i R_{v_i}$.

A useful accessing function called a projection allows the values of parameters to be obtained from the named set. It is defined from the entities into the range of a value set Vi. Formally, a projection is defined as

$$\text{proj}_{v_i} : S \rightarrow R_{v_i}.$$

As an example consider the assignment of a person's age and sex. A named structure is defined by the following

S = {Tom, Jane}

V = {age, sex}

R = { (age, [0, 130]), (sex, {male, female}) }

A = { (Tom, (23, male)), (Jane, (21, female)) }.

A projection function on the age parameter and an application of the function is given by

$$\text{proj}_{\text{age}} = \{ (\text{tom}, 23), (\text{Jane}, 21) \}$$
$$\text{proj}_{\text{age}}[\text{Tom}] = 23.$$

The projection function will be abbreviated in this work with the dot notation similar to typical programming languages.

$$\text{Tom.age} = 23 \quad \text{represents} \quad \text{proj}_{\text{age}}[\text{Tom}] = 23$$

The conceptual and pragmatic convenience of named sets is the basis of building a fact base for a knowledge-based system in hybrid model theory. For example, if the projection function is

considered to be a predicate, then the Prolog style predicate sex[Tom, male] and the equation Tom.sex = male can be considered equivalent. When reviewing example models which used graphical formalisms, it was found that labeling arcs and nodes with text was always performed. This is extremely valuable to humans during the development of a model. There was also a tendency to be fairly consistent with the usage of verbs and nouns on arcs and nodes. Since the interpretation of nodes and arcs in these formalisms is relatively straightforward (i.e., arcs and nodes have relatively well-defined semantics in each of the formalisms), the text is included as part of hybrid model theory by using named sets.

CHAPTER 3
FORMALISMS

Graph Theory

The formalisms presented in this chapter all have graphical equivalents. Since the graphs and the mathematical theory correspond to each other, the most convenient form will be used in the explication. In some of the formalisms, there is little or no distinction between graph and theory.

Although graph theory is typically not used as a modeling formalism, most formalisms use graphs as pictorial representations or equivalents. In a general sense, a graph of a system is an abstraction. It shows states, components, transitions, data flow, causality, etc. In most cases it does not show computational or analytical properties. Therefore, it is a simplification of a system.

Graphs are so useful as a modeling tool for humans that it would be imprudent to dismiss graph theory as a primitive theoretical modeling tool. It is assumed in this work that the structure of a graph is the defining factor for its usefulness as opposed to some physiological or psychological characteristic such as it is visual or pleasing to work with. Additionally, there are many analytical properties of graphs that are useful: spanning trees, articulation points, etc.

All formalisms used in this paper use directed multigraph representations. The directed property is used to indicate transition in state formalisms or data flow in procedural formalisms. The multi property is used to represent alternative next states in state formalisms and multiple data flow in procedural formalisms. Formally, a graph is a set of vertices V and set of edges $E \subseteq V \times V$. A nondeterministic system can easily be defined for state formalisms in which V is the set of states. This is, given a graph $G = (V, E)$ then a system $Z = (T, I, S, A, B, \delta)$, where

T is empty,

I is {{}},

S is V,

A is {{}},

B is a set with one element D defined as $\{(vl, n) : vl \in V$ and $n = \{v : (vl,v) \in E\}\}$, and

δ is $\delta(a,t) = D$ for all $a \in A$, $t \in T$.

This system starts in some nondetermined state, changes the current state nondeterministically by following some edge, and stops if it reaches a vertex with no edge leading out.

The graphical representation of a procedural formalism can be described by a system, but is is so simplistic (one state and no behavior) that it would be senseless to give the definition. Most derivations of system theories (outside of AI) are structured for analytical purposes at the very lowest level of abstraction (i.e., statistics). However, the graph of a system does represent information about the system causality, and yet it is rarely represented within traditional analytical techniques.

Finite Machines

Automata theory will be used to describe state machines. Differences will be pointed out when they occur. The typical use of automata theory is to model computational processes or analyze grammars. Most theoretical information in this section is derived from [Hop79].

The main objects of a finite state automata (FSA) are state, input, and transition. Given a particular state and input, the transition function dictates the next state. Automata transfer state until a final state is reached. A push down automaton (PDA) uses a stack which is a last-in-first-out storage device with unlimited capacity. A PDA transfers from state to state given the current state, current input, and the element on top of the stack. The next state includes the definition of a new stack top.

Figure 3.1 shows a typical PDA. The arc from s1 to s3 with label a/x indicates a transfer from state s1 to state s3 when input a is given and the element on the stack top is x. The initial condition for a FSA or PDA is the start state. It is part of the FSA or PDA's formal description. Unlike a system in system theory, an FSA or PDA has final states. This implies (and is most often the case) that the FSA or PDA will eventually stop when given valid input. However, in the most general automata, turing machines, this can not be guaranteed. If the labels of a PDA are extend to a/ ß, where ß is a string of symbols, then the automaton is called a context finite state automata (CFSA). The context is ß. The context can be used to store a history of the prior states or input. A CFSA can essentially examine the history of itself when deciding the next state.
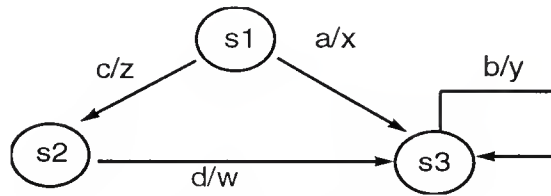

Figure 3.1 Example PDA.

An FSA is very similar to a discrete system. Only the definition is given here, but a similar definition and proof can be found in the literature [Wym76]. Given an FSA $= (Q, \Sigma, \Omega, s_0, F)$, where Q is the set of states, Σ is the input alphabet, Ω is the transition function, s_0 is the initial state, and F is the final states, an equivalent system is $Z = (T, I, S, A, B, \delta)$, where

T is $\mathfrak{I}$,

I is Σ,

S is Q,

A is all functions in Σ^n, $n \in \mathfrak{I}$ (any sequence of symbols from Σ).

B is $\{\Omega\}$, and

δ is $\delta(a,t) = \Omega$ for $a \in A$, $t \in \mathfrak{I}$.

A system theory description of a PDA requires that the states be defined as Q x D*, where D is the stack alphabet. The next state function is similarly changed.

Automata can be extend to encompass the property of nondeterminism. This is similar to but distinct from the notion of random or stochastic process. A nondeterministic FSA (NDFSA) allows for multiple transitions to be valid at the same time. Likewise, a nondeterministic PDA (NDPDA) or a CFSA (NDCFSA) allows multiple valid transitions. Although it has been shown that FSAs are equivalent to NDFSAs and CFSAs are equivalent to NDCFSAs, the nondeterministic counterparts are usually more concise descriptions and are used more often.

There are many analytical properties about automata, especially FSAs and PDAs. Many of these properties analyze the class of languages that an automaton accepts. These properties are useful in the discussion of grammars, but they have not appeared in any literature relating to system theory and signals. The main interest in automata in system theory literature has been for control purposes and not for validating correct sequences of input.

The stack of a PDA or the context of a CFSA allows the system to have a memory. This is a very distinct concept from any other formalism in this chapter and from classical system theory. Although the notion of memory and internal state are highly related formally, the difference in meaning can play a major role when attempting to interpret the system.

Time is a notion easily integrated within automata. However, there is a difficulty when multiple formalisms are used. In a system representation of an automata, time is actually used to number or sequence the input. This works for descriptive purposes, but when integrating this with a continuous system where the time is $\Re$, there is still no specific identifiable relationship.

The notion of steady state, which is important in system theory and several of the other formalisms in this chapter, is not pertinent for any of the automata. A nonterminating automaton is undesirable. For some formalisms (to be presented later), nonterminating behavior is essential for many analytical properties. This does not involve many theoretical difficulties, but introduces disparate images of the purpose of a system constructed from multiple formalisms.

Markov Systems

A Markov system represents a stochastic process. It is a state representation. The transitions in a Markov system are stochastic. Each transition probability is based on the assumption that any past or future state is conditionally independent given the present state.

Formally, a Markov system is a pair, $Z = (S, T)$ where

S is a finite set of states, and

T is a function $f:S \times S \rightarrow [0.0, 1.0]$ called the conditional probability.

The conditional probability $T(s_i, s_j)$ represents the probability of the next state s_j given the current state s_i. In probability theory this is $T(s_i, s_j) = p(\text{next state} = s_j \mid \text{current state} = s_i)$. It is required of the function T that

for each state s_i $\quad \Sigma_k T(s_i, s_k) = 1$.

This stipulates that the probability of the next state transitions add up to one. Figure 3.2 shows a simple Markov system with the transition probabilities on the arcs.

A sequence of states $s_i \ldots s_k$ is called a Markov chain. The initial state of a Markov chain can be given in several ways. The conditionally probability function $T(s_i, s_k)$ is usually represented as matrix M. An entry in row i and column k is the probability $T(s_i, s_k)$. It can be shown that the probability of being in state s_k in the chain $s_i \ldots s_k$ is $\delta(s_i)(M^n)_{i,k}$, where n is the length of the chain, M^n is the nth product of M with itself and $\delta(s_i)$ is the initial probability of s_i [Cly90].
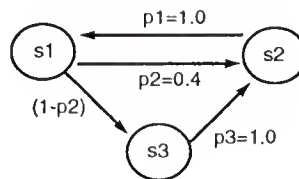

Figure 3.2 Markov System

If there exists an n such that no entry in the matrix M^n is zero, then the Markov system is called regular. In a regular Markov system it is possible to go from a state to any other state in no more than n transitions. It can be shown that the limit as $n \rightarrow \infty$ will reach steady state equilibrium probabilities. If $SS(s_i)$ is the steady state equilibrium and $(M^n)_{i,j}$ is the entry row i and column j of the matrix M^n, then formally,

for any i, $\quad Lim_{n \rightarrow \infty} (M^n)_{i,j} = SS(s_i)$.

In comparison to system theory, there are several apparent differences between the two types of systems. A Markov system has no input. The environment can not affect the state of a Markov system. If it could, then many of the analytical properties would be unsound. Additionally, the initial state of a Markov system is nondeterministic. In the context of other formalisms, a way to specify the initial state must developed. For instance, if a Markov system is a submodel of a state within an FSA, then when that FSA state becomes active, the Markov system must be initialized. This can be nondeterministic; however, a deterministic method could also be devised.

Since a Markov system has distinct states, it can be classified as a discrete-event model. An event in a Markov system is the selection of a transition. As a separate formalism, this does not complicate a system theoretic description of the Markov system. When using several formalisms together, this does create a problem. The events of different formalisms must be related to each other in some manner. Unfortunately, there is no time associated with transitions in a Markov system. If a state in a Markov system is submodeled with a formalism which explicitly uses time, then how do the other Markov states events relate to this submodel?

A Markov system does not have a transition function. It is possible to develop one similar to a NDFSA. However, a way to assign a probability to the next state based on the conditional probability instead of the input must be developed.

The behavior functions of a Markov system correspond to all possible finite paths through the graph. Each of these paths has a probability associated with it. Typically, the steady state

equilibrium probabilities of a Markov system are considered to be the defining factor in describing that system.

Petri Nets

Petri nets are typically used to model concurrent systems in which the objects of the system must have synchronized behavior. Additionally, Petri nets can be used to model resource allocation systems. The source for the information about Petri nets in this section is derived form Peterson's book [Pet81].

There are three main objects in Petri nets: places, transitions, and tokens. Places and transitions alternate nodes in a graph (see Figure 3.3). A transition moves a token in an input place to an output place. This movement is called firing the transition. These attributes categorize a Petri net graph as a bipartite directed multigraph. The state of a Petri net is the number of tokens in each place. There is no time associated with Petri nets. Transitions fire, nondeterministically, transferring tokens form place to place. After a transition has had a chance to fire, the Petri net is said to be in a new state.
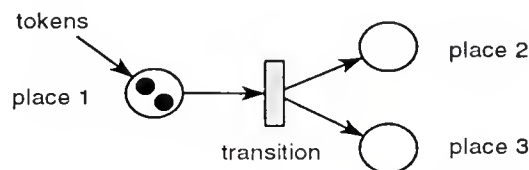
Figure 3.3 Example Petri Net

A Petri net is defined formally as a 4-tuple, $Z = (P, T, I, O)$ where

P is a finite set of places,

T is a finite set of transitions,

In is a function $T \rightarrow P^{\infty}$ called the input function, and

Out is a function $T \rightarrow P^{\infty}$ called the output function.

Since P is a finite set, it is clear that P^∞ is not a set but a bag (a set which allows duplicate values). Since a transition can have multiple arcs to the same output place, multiple copies of places are allowed in the sets In(t) or Out(t). Theories of bags have been studied; however, it is just as easy to represent a bag {a,a,b,b,b,c} as the set {(a,2), (b,3), (c,1)}. In any case, the function #(In(t_j), p_i) retrieves the number of arcs from place p_i to transition t_j and #(Out(t_j), p_i) retrieves the number of arcs from transition t_j to place p_i.

A marking m:P $\rightarrow$ $\mathfrak{S}^n$ for a Petri net is a n-tuple where n = cardinality of P (|P|). A Petri net with 5 places and a marking m=(1,3,1,5,4) has 1 token in place 1, 3 tokens in place 2, 1 token in place 3, etc. The function m(p_i) retrieves the marking for place pi (i.e., above m(p_2) = 3). Markings in Petri nets and states in systems are equivalent.

Given a marked Petri net Z with marking m, a transition t in Z is said to be enabled if

for all pi   #(In(t), p_i) <= m(p_i).

A enabled transition means that the transition can get a token from each input place for each arc. For example, if there are two arcs from a place to a transition, then there must be as least two tokens in that place in order for the above condition to be met for that place. When a transition fires, the tokens are removed from the places. If two transitions require the same token from a place in order to fire, then only one will fire if there are not enough tokens for both transitions. The choice in this case is arbitrary (nondeterministic).

The next state of a Petri net is defined if at least one transition can fire. Otherwise the Petri net is blocked. Given a Petri net Z, a marking m, and a transition t, the next state δ(m,t) is formally defined as

δ(m,t) = m' where m'(p_i) = m(p_i) - #(In(t), p_i) + #(Out(t), p_i).

There are several differences between a Petri net formalism and systems theory. A Petri net has no input. When using a Petri net, an initial marking m_0 is given and next states are reached by firing transitions. The initial marking is a state not an input. Therefore, the environment can not effect the state of a Petri net.

Since a Petri net has distinct states, it can be classified as a discrete-event formalism. An event in a Petri net is the firing of a transition. However, there is no time associated with these events. As an independent formalism, this does not complicate a system theoretic description of a Petri net. When using multiple formalisms together, this does create a problem.

The events of different formalisms must be related to each other in some manner. An obvious choice would be to use time. With state machines, Markov systems, and Petri nets combined, the concept of time being related to state transitions (events) seems appropriate, but not the only possibility.

The transition functions of Petri nets and systems are similar. The behavior functions of a system roughly correspond to the sequences of allowable states. Since Petri nets are nondeterministic, the requirement that the system behavior be functions must be relaxed; a system's behavior is a set of relationships.

There are several important analytical characteristics which are important in Petri net theory. All of these have definitions, but only the concepts will be presented here. The reachability of a Petri net is a set of markings which are reachable from some initial marking. This set is designated as $R(Z, m_0)$. Safeness of a Petri net with a given initial marking is defined when all places have 0 or 1 token. A place is k-safe (k-bounded) if the number of tokens never exceeds k for some given initial marking. A marked Petri net which has a transition that can never fire is said to be in deadlock. The transition (or transitions) is said to be dead. Transitions which can potentially fire are called live.

Queuing Networks

One of the most common techniques used in simulation is queuing theory. It is used to model waiting lines. Banks or grocery stores are typical examples of queuing systems. Queuing

networks are process oriented. There are three main objects in queuing networks: populations,

queues, and servers. For analytical purposes queuing networks are conveniently represented by

attributes of these objects. The notation is m/n/o/p/q where m is the interarrival time distribution,

n is the service time distribution, o is the number of parallel servers, p is the system capacity, and

q is the queue discipline. The theoretical material in this section is derived from [Ban84] and
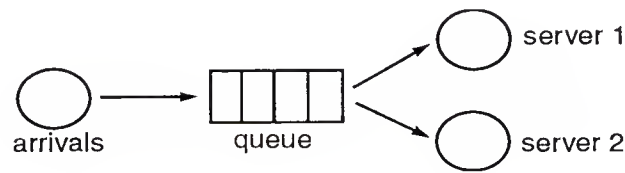
[Gra80].

Figure 3.4 Simple Queuing Network

Figure 3.4 shows a simple queuing network. The arrivals node represents the calling

population. This could represent customers at a bank, palettes in a factory, cars at a traffic light,

or calls at a telephone exchange. An arrival from the population is called a customer. The arrival

rate of customers depends on the population type. A finite population has a limited number of

customers. Therefore, the arrival rate is influenced by the number of customers currently in the

system. The arrival rate of customers for an infinite population is described by a distribution

(usually the Poisson distribution). When customers arrive in the system they are queued (wait in

line) until a server is available. In Figure 3.4 there are two server nodes. Each node represents 1

server. The service time for each server may be constant or random. Common random service

times are represented by the exponential, gamma, and normal distributions.

The queue node in Figure 3.4 appears to be a first-in-first-out (FIFO) line. This is a

consequence of the graphical portrayal and not a true depiction of the queue's discipline. The

same graphical description may represent a FIFO, last-in-first-out (LIFO), priority (PRI), or any

other queue discipline. The capacity of a queue is the space available in the queue. If the queue's

capacity is exceeded, then the system balks; arriving customers leave the system without entering

the queue. It is also possible for customers to leave the system if they wait to long or change queues in a multiqueue system.

A queuing network with a Poisson arrival distribution has an exponential interarrival time [Gra80]. If the service time for the servers in Figure 3.4 is exponential, the arrival rate is exponential, the queue's capacity is 5, and discipline is LIFO, then the notation used to represent the system is M/M/2/5/LIFO (where M stands for an exponential distribution). However, a complex system with multiple populations, queues and servers cannot be described with this notation. Most real world models are so complex that no attempt is made to a use concise notation; instead, the graphical representation is used.

There have been many analytical properties developed for simple queuing networks. Since the variety of networks is very broad, each network type has different theoretical derivations. There are, however, common properties among the different types of networks. The distributions used to describe the arrival and service times are examples. Because the analysis is so varied, only a verbal definition of the most useful properties is presented.

The expected time in the system is the average time a customer spends in the system. The time a customer is in the system is comprised of the time in queues and the time being served. Although it is typical to keep the expected time in the system as small as possible, the ratio of time waiting to time in the system is more important (especially to a customer). The expected number of customers in the system is important for determining queue capacities or the number of servers required. It is undesirable to have customers balk because queues are to small or to have servers idle because there are to many servers. The expected queue length and server utilization are measurements which help in deciding the system's type as described above. The throughput of a system measures the number of customers served per unit time. Many times, the objectives of a queuing system are opposed to each other; for instance, to maximize throughput and utilization while minimizing waiting and service times.

When a particular queuing network fails to meet the requirements of a population (determined by one or more of the above measurements), alternative types of networks are investigated. The

most complex of which is to change the queuing discipline (as opposed to changing the number of servers or increasing the capacity of a queue). This has lead to a multitude of queue disciplines. Additionally, more realistic parameters have been introduced (balking, changing lines) or more complex networks. Because these are very difficult (if not impossible) to analyze, numerical approximations have become a norm in the analysis of complex queuing networks.

The computational methods used in queuing networks are usually extensions to the above material. For example, customer attributes, resources and macro models are available in many of the commercial packages such as GPSS [Sch91] and SIMAN [Peg90]. Additionally, more sophisticated packages have animation, such as CINEMA [Kal91], or graphical entry methods, TESS [Sta87]. Despite this, the basic tenets for these system have evolved out of queuing theory.

In terms of system theory, a queuing network is a discrete-event, nondeterministic process. The state of which can be described by a tuple. Each server and queue have a position in the tuple. For instance, a 2 queue, 2 server network is described by (q_1, q_2, s_1, s_2) where q_1 and q_2 are the number of customers in the queue. The servers s_1 and s_2 are either idle (0) or busy (1). The time base is $\Re$ and there is no input into the system. The behavior can only be described in terms of steady state.

Control Theory

Control theory is based on linear system theory. The types of systems modeled in control theory are continuous systems. Therefore, a direct description using the systems approach is easily obtained. The system description is not presented in this section, but can be found in many books [Wym77, Zei86, Dor86]. What is of interest is the cause-effect relationship embodied in control theory. Although classical control theory does not include these relationships directly, other methods, such as bond graphs [Tho75], have shown that the cause-effect relationship plays a role in the designer's conception of the system.
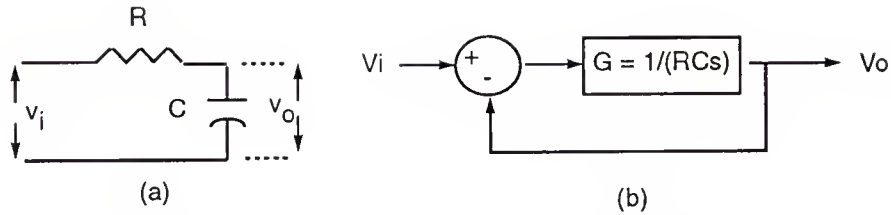
Figure 3.5 (a) Low Pass Circuit (b) Block Diagram

Typically, linear systems are described with the aid of block diagrams. Figure 3.5 exemplifies the cause-effect relationship of a low pass RC circuit (integrating circuit). Using Control theory, the transfer function $V_0(s) / V_i(s)$ can be obtained. The input voltage $v_i(t)$ and the output voltage $v_0(t)$ are

$$v_i(t) = i(t)R + 1/C \int i(t) \, dt \quad \text{and} \quad v_0(t) = 1/C \int i(t) \, dt.$$

The Laplace transforms are

$$V_i(s) = I(s)R + 1/(Cs) \, I(s) \quad \text{and} \quad V_0(s) = 1/(Cs) \, I(s).$$

Solving for $I(s)$ in the equation for $V_0(s)$ and substituting into the equation for $V_i(s)$, the transfer function is

$$V_0(s)/ V_i(s) = 1/(RCs + 1).$$

The transfer function in the form of a block diagram is shown in Figure 3.5. The block G is an integrating block. The diagram indicates the qualitative functioning of the circuit. At time t, the feedback of the output voltage is subtracted with the input voltage and then integrated to yield the output voltage at time t+dt. If the input voltage is constant, then it is easily deduced that the

output voltage becomes constant after some finite time. This is exactly what a low pass circuit does under constant voltage. The qualitative cause-effect relationship was deduced in a domain independent manner. This type of reasoning is the focus of both AI research [Bob86] and knowledge-based simulation [Fis91b].

The state of the low pass circuit in classical systems theory could be represented by the tuple $(v_i(t), v_o(t))$. Intuitively, however, there are two states of the circuit when the input voltage is constant: the transient state (when the capacitor is charging) and the steady state. Control theory does not encompass this abstract notion of state. However, the block diagram's graph indicates indirectly the existence of these states.

Although control theory has a simple description in systems theory, integrating continuous formalisms with discrete formalisms requires resolving the time base. There are only two possible resolutions: discretize the continuous system or assign time values to the discrete system.

A linear system is an important concept in control theory. In Petri nets and state machines linearity is not discussed (at least not in the literature found to this date). A linear system exhibits two essential properties: superposition and homogeneity. Given a system that with input $x(t)$ produces output $y(t)$ and with input $w(t)$ produces output $z(t)$, if input $x(t) + w(t)$ produces output $y(t) + w(t)$, then the system has the superposition property. If input $Bx(t)$ produces output $By(t)$, where B is a constant, then the system has the property of homogeneity.

Control theory, along with the other four formalisms presented in this chapter, all have a system theoretic description. In the next chapter, a theory will be introduced which provides the foundation for a consistent and cohesive mechanism to describe these formalisms and will permit them to be used in a hierarchical organization.

CHAPTER 4
HYBRID MODEL THEORY

Model Structure

Hybrid model theory is a combination of general system theory (GST), named sets, and graph theory. The combination of GST and named sets is also used in multifaceted modeling [Zei84]. In a sense, one could argue that hybrid model theory is a derivation of GST. Mathematically, this may be correct. However, there are significant differences of which those familiar with GST should be aware. These differences arose from the requirements needed for HH modeling .

First, hybrid model theory is not to be used by the investigator. It is not a modeling formalism. It is a generalized formalism that is not as conceptually efficient as formalisms such as Petri nets, queuing networks, and block diagrams. Second, although building bottom-up is possible, hybrid model theory emphasizes a top-down approach to constructing a model. The idea behind hybrid model theory is to take a model which is partially correct in describing a system's behavior and refine only those components which do not correspond with observed data or system specifications. Third, hybrid model theory focuses on the analysis of a single system under development. HH modeling and hybrid model theory are meant to provide the foundation for a computer environment which allows for the creation and investigation of system models, not the classification, identification, comparison and retrieval of already constructed and understood system models. Hybrid model theory deals with the alteration and investigation of incomplete or incorrect models.

With this in mind, the definition of a model in hybrid model theory is introduced. A model M is a named set such that $M = <H, A, X, \Psi, \Theta, \tau, \beta, \delta, \mu, \lambda>$ and

H :Component	<self , η_1, ...>
A : Edge	<α_1, α_2, ...>
X : Input	<χ_1, χ_2, ...>
Ψ : Output	<ψ_1, ψ_2, ...>
Θ : State	<...>
τ : Time Domain	<(T,+,$\leq$), zero, delta, map[], magnitude[]>
β : Initialize function	(T,C,Q) -> θ
δ : Transition function	(T,C,Q) -> θ
λ : Output Function	(T,C,Q) -> ψ.

The symbols < and > indicate the use of named sets, and elements in these sets can be accessed as described in Chapter 2. The component set (H) of a model always has the special symbol self as a member. This symbol is used to indicate a reference to a submodel (if one exists). For most atomic models the self symbol is the only member of the component set. In structured models, the component set contains the models which are supervised by the controller model. The edge set (A) of an atomic model is empty. In structured models, the connectivity between component models is identified with the edge set. An edge $\alpha \in A$ is a named set of the form <to, from, type>, where to and from are models in the component set (H) and type is either † (undefined), a standard data type ($\mathfrak{R}$, $\mathfrak{I}$) or a model. Together, the components and edges describe the graph of the model and either what type of data is passed between the components or how control is transferred among the components.

In Chapter 2, a brief description of intermodel coordination (coupling) was represented in hybrid model theory. From the definition above, it can be seen that a structured model, which has components that are also structured models, represents intermodel coordination. The root model is a group controller model. It controls the parallel operation of structured models. In intramodel coordination, a structured model coordinates atomic models. The distinction between inter and intramodel coordination appears to be just conceptual; however, the structured models that

coordinate the atomic models (state, parallel, and selective controllers) have a very different form and semantics from the group controller that coordinates a set of structured models. It is the distinct form of controllers that allows heterogeneous refinement.

The input (X) and output (Ψ) also have the form <to, from, type>. These sets signify the data or control information used by different types of models. For models in which the input has not yet been specified, the from model will be equal to † (undefined). The same definition also applies for a model's output. The only difference between inputs and outputs are that inputs are signals (functions over time) and outputs are values. The state (Θ) named set is used for a variety of purposes. It is very similar to local memory in computational definitions. It can contain any other type of named set (including a model). However, for analytical purposes, the state set should contain only constants or functions of time.

The time domain of a model was discussed in Chapter 2. It is only noted here that the time domain of a model can be null. However, it is intended that models which do not have the notion of time in the clock sense include notions of time in the computational sense. That is, if a model is not measured in seconds, but has a definite sequence of computation, then the model should use an integer time domain. Each integer X+1 represents the next computational step.

The last three elements of a model are functions. Typically, these are used to compute the new state and output trajectories over a time interval. Because hybrid model theory is centered around simulation concepts, these functions have been conceptually altered. It is assumed that all three functions use two times: the current time (a global variable) and an input time (given at function invocation). These times are used to calculate the state or output at the input time. The current input and state are also assumed to be part of the input to these functions. Trajectories are created by symbolic methods which take a model as input or created through numerical techniques. Additionally, it should be emphasized that these functions are declared, not precompiled. When numerical analysis (simulation) is needed, the declarative model can be compiled and optimized (unless an interpretative language like LISP or an object-oriented language is used).

The initialization function (β) is necessary since models can be dynamic. At any time during analysis, a model can become active. This not only allows for the modeling of systems which may lie dormant, but more importantly, it models systems which have multiple descriptions over time. A piecewise continuous system is an example of a primitive multidescription system. In this work, the state oriented formalisms implement the piecewise concept. For data flow models (differential equation models), the initialization function (β) sets the initial conditions.

The transition (δ) function is intended to be used when a model is active. Although, as can be seen from the description, it could be used to initialize a model. The initialization and transition functions were derived so that the concept of state, transition and initialization could be separated. Again, this is necessary in symbolic and interpretation methods. If the transition function uses stochastic or nondeterministic relationships, then mathematically the term function is incorrect; however, for purposes of this discussion, the term function will be used in a similar manner to that used in programming languages.

For the same reason (and tradition), the output function (λ) is also kept separate from the other functions. One of the optimizations for numerical analysis is the integration of these functions so that only one call to the model produces the total behavior. This integration is possible in hybrid model theory because there are only four controller models and each type of controller has the same form of transition and output functions.
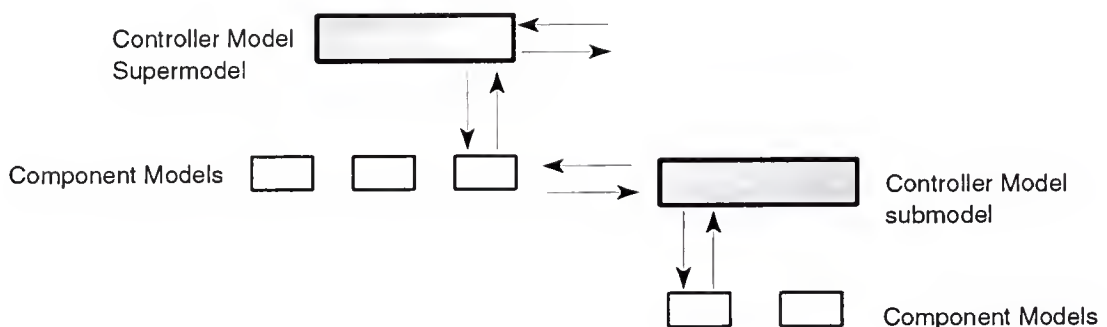
Figure 8   Signal Flow in Intramodel Coordination

There are two rules which capture the manner in which atomic components of a structured model can be coordinated with other models (intramodel coordination). This is a pseudo formal definition of intramodel coordination in hybrid model theory. Figure 8 shows how the coordination is accomplished. The rules can be stated as follows.

1. A component (node) in a model can have its operation's output delayed or altered by another model called the submodel. The component model, when activated, initializes the submodel and waits for a signal of completion, which then deactivates the submodel.

2. A component (node) in a model can have its operation replaced by another model; however, the I/O and control of the submodel must be the same, the model is continuous over the analysis.

The types of controllers which interact with each other dictate which of these rules applies when submodeling. The higher level model in submodeling is called a <u>supermodel</u>. Like most model formalisms, the arrows in Figure 8 signify control and data flow between model types. The rule which applies depends only on the supermodel involved. For instance, if the supermodel is a state controller, then rule 1 above applies, for a parallel supermodel only rule 2 applies and for a selection supermodel rule 1 applies.

State Modeling

A <u>state controller</u> model manages a finite set of components. The edges represent control flow between the components. The state controller model is responsible for the input and output to the current state model. A <u>state</u> model represents the individual components. For example, in a Markov system, the state controller is the controller model and the state models are the component models of the controller model. The components of a Markov system do not use input.

However, each Markov state can supply an output at any given time as described in Chapter 3. The input to a Markov state is only used by the submodel (if one exists).

A state machine can be represented in the same manner as a Markov system. The only difference is in the internal state of the state models. Hence, in order to describe state formalisms, three models are needed: two atomic models and one structured model. Specifically, these are a finite state atomic model, a Markov state atomic model and the state controller structured model. The atomic models are defined first; then, the state controller is defined.

A finite state atomic model is defined as

H :Component	<self>
A : Edge	Ø
X : Input	$<\chi_1, \chi_2, ...>$
Ψ : Output	$<\psi_1, \psi_2, ...>$
Θ : State	<Table, nextState>
τ : Time Domain	<(T,+,≤), zero, delta, map[], magnitude[]>
β : Initialize function	(T,X,Θ) -> <Table, initialState>
δ : Transition function	(T,X,Θ) -> <Table, nextState>
λ : Output Function	(T,X,Θ) -> Ψ.

Almost all atomic models have only the symbol self as a member of the component set and have no edges. The input is identical to the supermodel's input (X). Although atomic models can be used by themselves, it is intended that all atomic models be part of a controller model. The state (Θ) of a finite state model consists of the next state and a function defined as

Table(τ,X) -> S, where S is the Component set of the state controller model.

Thus, the table is a function which returns the next state model of the controller model given the current input and time. The function takes the form of if-then statements. Typically, a finite state

machine has a single table. Given the current input, the next state is looked up in the table. In hybrid model theory, each state has its own table. This is necessary for intramodel coordination. The output types of the state model are the same as the controller model's output.

The other sets of a state model are fairly simple. The time domain typically matches the time domain of the state model's controller model. The output function can be any function from the input, time and state into the outputs Ψ, given the above conditions. The transfer function uses the table to formulate the next state as

$$\delta[\tau,\chi,\theta] = <\theta.\text{Table}, \theta.\text{Table}[\text{map}[\tau],\chi]>.$$

A Markov state model is the same a state model except for the table function. A finite state model table function might have entries of the logical form "if (input1 = 4.0) then nextState = self." A Markov state model would have entries of the logical form "if (currentProb < 0.3) and (currentProb > 0.5) then nextState = self." The currentProb variable is a state variable generated by the controller model at specific time intervals.

In order to coordinate the atomic models, a state controller must maintain all the information necessary to change state and must direct input and output to and from the state models it controls. A state controller is defined as

H :Component	$<\text{self}, \eta_1, ...>$
A : Edge	$<\alpha_1, \alpha_2, ...>$
X : Input	$<\chi_1, \chi_2, ...>$
Ψ : Output	$<\psi_1, \psi_2, ...>$
Θ : State	$<\text{currentState}>$
τ : Time Domain	$<(T,+,\leq), \text{zero, delta, map[], magnitude[]}>$
β : Initialize function	$(T,X,\Theta) \to <\eta_i>$
δ : Transition function	$\text{Tr}[\tau,\chi,\theta]$
λ : Output Function	$\text{Out}[\tau,\chi,\theta].$

The component set contains the state models. The edge set contains the arcs indicating the control flow from one state to another. The edge set represents a declarative form of the table functions for all the states (components) of the model. The input can be any type of signal as typically defined in system theory. The state controller has only one internal variable in the state set (Θ). This is the <u>nextState</u> model. The initialization function gives the starting state. The transition and output functions both have very similar forms. For instance, the transition function $Tr[\tau,\chi,\theta]$ is defined as

$Tr[\tau,\chi,\theta] = RTr[currentTime, \tau, \chi,\theta]$ where

$RTr[t1,\tau,\chi,\theta] =$

    if (map[t1] <= map[τ]) $RTr[t1+dt, \tau, \chi,<\theta.currentState.Tr[t1+dt,\chi,\theta].nextState>]$

    else $<\theta>$.

The $Tr[]$ function is actually a pair of functions. The $Tr[]$ function sets up the conditions for the recursive function $RTr[]$. The $RTr[]$ function recurses on itself while $t1 < \tau$. On each recursion $t1$ is incremented by dt, the delta time of the model. Therefore, numerically, the transition function is a while loop which continues from the current time until the end time by increments of dt. During each loop the current state model's transition function is invoked. It returns the next state for that time period. This continues until the state at the end time (τ) is returned. Also, note that the times are converted by the time domain function map[t1].

The output function $Out[\tau,\chi,\theta]$ has the exact same form as the transition function except the the outputs are retrieved and the $ROut[]$ function is invocated; that is,

$Out[\tau,\chi,\theta] = ROut[currentTime, \tau, \chi,\theta]$ where

$ROut[t1,\tau,\chi,\theta] =$

    if (map[t1] <= map[τ]) $ROut[t1+dt, \tau, \chi, <\theta.currentState.Out[t1+dt,\chi,\theta].nextState>]$

    else $\theta.currentState.Out[t1,\chi,\theta]$.

For computational purposes, the transition and output functions can be combined. This function is designated as Comp[]. For a state controllers, the following pseudo code demonstrates the computation of the next state and output at time τ, given the current time is declared in the variable currentTime.

```
initialize();
tempTime = currentTime;
While   (tempTime < τ)
     output = currentState.Comp[tempTime, input].output;
     currentState = currentState.Comp[tempTime, input]. nextState;
     tempTime = tempTime +deltaT;
```

The current state's Comp[] function is invocated to determine the current output and the next state. This controller works for any type of state model. For state machines, the state model encapsulated the behavior of state machines. For Markov systems, the Markov state model encapsulated the behavior of the Markov system. However, the basic features which make all state formalisms similar is captured in the state controller model.

From these definitions, intramodel coordination can easily be formulated for state formalisms. Any model type that has input and output which match the state model can be used as a component in the state controller model. The same is also true for a Markov system. The only constraint is given by rule 1 stated in the first section. It is repeated here and modified to suit state modeling.

1. A state model in a state controller model can have its operation's output delayed or altered by another model called the submodel. The state model, when activated, initializes the submodel and waits for a signal of completion which then deactivates the submodel.

Each state in a state machine can have a submodel (a refinement of the state model). The state model is an abstraction of the submodel. When the state become active, the submodel computes the current output. However, the decision as to what is the next state is still controlled by the state model.
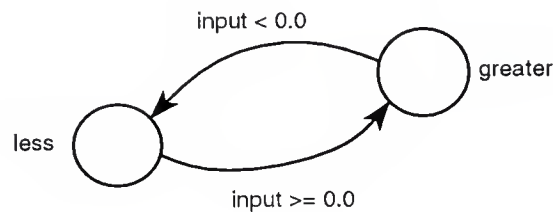

Figure 4.3 Simple State Machine.

As an Example, the operational definition of the state machine in Figure 4.3 is given. There is one input and one output signal.

```
stateMachine[startTime, endTime, input]  {
    currentState = less;
    tempTime = startTime;
    while (tempTime < endTime)
     {switch (currentState) {
        case less:    output = 0;
                      if (input >= 0.0) currentState = greater;
                      break;
        case greater: output = 1;
                      if (input < 0.0) currentState = less;
                      break;
      };
     tempTime = tempTime + deltaT;
    };
    return output;
}
```

If the input is less that zero the output is zero. If the input is greater than or equal to zero, the output is 1. It is reemphasized, that the operational definition is compiled strictly from the model

definition. Thus, it is a completely automated procedure. If the less state had a submodel, then the only difference would be the assignment of the output. Instead of "output = 0," an invocation of the submodel would be called as "output = subModel[tempTime, tempTime+deltaT, input]." The submodel could be, for instance, another state controller.
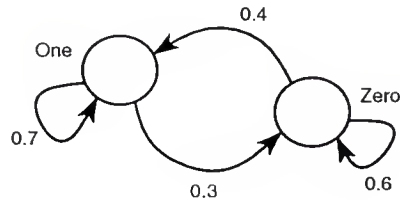

Figure 4.4 Markov System

The operational definition of a Markov system is very similar. There is one input and one output signal. In Figure 4.4 a simple Markov system that outputs zero in stateZero and one in stateOne is shown. The operational definition is as follows:

```
markovSystem[startTime, endTime, input]  {
    currentProb = random[];
    tempTime = startTime;
    currentProb = random[];
    while (tempTime < endTime)
      {switch (currentState) {
        case Zero:   output = 1;
                     if between(currentProb, 0.0, 0.4)  currentState = One;
                     break;
        case One:    output = 0;
                     if between(currentProb, 0.0,0.3)   currentState = Zero;
                     break;
      };
      tempTime = tempTime + deltaT;
      currentProb = random[];
      };
    return output;}.
```

Parallel Modeling

A parallel controller model manages component models which operate in parallel. The edges of a parallel controller model represent data flow. This is similar to a data flow diagram; however, data flow is typically sequential. A parallel controller can model sequential data flow, but there would be transient states between changes in output.

A parallel controller is essentially the same modeling paradigm as general system theory. Hybrid model theory ensures that this paradigm can be integrated with heterogeneous model types in a hierarchical manner. Additionally, the parallel controller concept is a generalization of parallel data flow models just as a state controller was a generalization of state models. The component models which a parallel controller model manages are functional models. There are, for each modeling formalism, only a finite number of functional models. In data block diagrams (control theory) there are 5 types: integral, derivative, summation, generic, and multiply.

The definition of the functional models is trivial. However, as an example, the definition of an integral function is given here.

An integral model is defined as

H :Component	<self>
A : Edge	$\emptyset$
X : Input	<input=<to,from,R>>
Ψ : Output	<output=<to,from,R>>
Θ : State	<sum=<R>>
τ : Time Domain	<(T$\Re$,+,$\leq$), zero, delta, map[], magnitude[]>
β : Initialize function	$\beta[\tau,\chi,\theta]$ = <k>, k a constant
δ : Transition function	$\delta[\tau,\chi,\theta]$ = <(χ.input[τ] * deltaT) + $\delta[\tau-deltaT,\chi,\theta]$>
λ : Output Function	$\lambda[\tau,\chi,\theta]$ = $\delta[\tau,\chi,\theta]$.sum.

Because an integral model is atomic, the component set only contains self and the edge set is empty. There is one input and one output. The state contains the sum of the integration up to the

current point in time. The initialization function initializes the integration sum. The transition and output function are essentially identical. The output of the current integration is the sum used for the next integration step. Although it appears that this is only a numerical approximation to integration, the transition and output functions are used only for simulation and reasoning methods. Symbolic routines do not need to use the transition function. A symbolic routine only needs to identify the <u>type</u> of atomic model. In Chapter 5, an example is given which demonstrates this point. A special case of this is the generic functional atomic model which can be used to model functions such as sin(), cos(), exp(), etc.

The parallel controller manages all the data input and output between the functional atomic models in its component set. The edge set dictates the input/output relationships. A parallel controller can be defined as

H :Component	$<$self , η_1, ...$>$
A : Edge	$<\alpha_1, \alpha_2, ...>$
X : Input	$<\chi_1, \chi_2, ...>$
Ψ : Output	$<\psi_1, \psi_2, ...>$
Θ : State	$<<$last, next$>$, $<$last,next$>$, ...$>$ for each edge
τ : Time Domain	$<(T_{\Re},+,\leq)$, zero, delta, map[], magnitude[]$>$
β : Initialize function	Init$[\tau,\chi,\theta] = <<$last, next$>$, $<$last,next$>$, ...$>$ for each edge
δ : Transition function	Tr$[\tau,\chi,\theta]$
λ : Output Function	Out$[\tau,\chi,\theta]$.

The component set contains the functional models (the nodes in the graph of the block diagram). The edge set contains the connectivity. The state set has an entry for each edge. The form of the entry is $<$last, next$>$. Conceptually, at each instance in time, the output from each functional component is stored in the parallel controller state. The input for each functional component is also retrieved from the parallel controller state set. Optimally, for instance in an numerical

implementation of the parallel controller, these temporary variables are eliminated. The

initialization function produces the initial inputs and outputs by initializing the state set.

The transition function Tr[] takes the current state and invokes the functional components

transition functions to produce the next state. The definition of TR[] is

$Tr[\tau,\chi,\theta] = RTr[currentTime, \tau, \chi,\theta]$ where

$RTr[t1,\tau,\chi,\theta] =$

    if $(t1 <= \tau)$ $R2Tr[H,t1,\tau,\chi,\theta]$

    else $<\theta>$

$R2Tr[H,t1,\tau,\chi,\theta] = RTr[H,(t1+deltaT),\tau,\chi, \theta']$

where $\theta' = <s_1, s_2,...>$

and $s_i = <\theta.\eta_1.next, H.\eta_1.Tr[t1,\theta.\eta_1.last, H.\eta_1.\theta]>$.

This transition function is similar to the transition function for state controllers. The first

function is recursive and handles the increments in time, just as in the state controller model. The

second function constructs at each time increment the current input and output of the functional

atomic models. The new state θ' is constructed in this manner. Each entry of the state θ' is an

invocation of the functional components. The output function is constructed the same way;

however, instead of producing a new state, the current output is produced.

Intramodel coordination is accomplished exactly the same way as in the state controller. This

is the objective of hybrid model theory. Any functional component of the parallel controller can

be submodeled. However, only the generic atomic models need to be refined since the others

perform specific functions. As with the state controller, any model which has the same input and

output of the generic functional model can be used in place of the generic model. For instance, a

state machine can compute the input/output relationship of a generic function in a block diagram.

The parallel controller of the block diagram would still operate the same. Hence, the state

machine has been coordinated with a component in the block diagram.

There is one difference between the coordination of the state controller and the parallel controller. With the parallel controller rule 2, as stated in section one, is used. This is,

2. a functional model in a parallel model can have its operation replaced by another model; however, the I/O and control of the submodel must be the same, the model is continuous over the analysis.
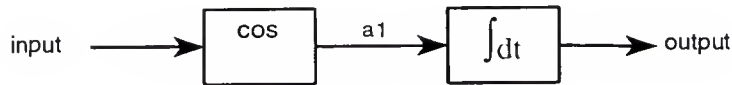

input → cos — a1 → $\int dt$ → output

Figure 4.5 Block Diagram

As with the state controller, an operational definition of the parallel controller is given for an example. Figure 4.5 is a simple block diagram for the equation output = $\int_t$ cos(input) dt. The operational definition for this is

```
blockDiagram[startTime, endTime, input] {
    initialize[];
    tempTime = startTime;
    while (tempTime < endTime)
     {
       a1.next = cos(input);
       output = integral[tempTime, tempTime+dettaT, a1.last]
       a1.last = a1.next;
       tempTime = tempTime + deltaT;
     };
    return output;
}.
```

From the operational definition, the intramodel coordination can be clearly demonstrated. If the cosine functional block were submodeled with another model, the a1.next state variable at

any instance in time would be calculated by the submodel. In addition, the form of the operational definition for the state controller and the parallel controller are very similar. This is a result of hybrid model theory.

Selective Modeling

The selective controller is the most complicated type of controller. The selective controller is also a parallel controller, but there are two distinct types of atomic models, and there is a nondeterministic choice of allocating data. The two atomic models are <u>functional</u> and <u>storage</u>. For example, the places in a Petri net are storage models and the transitions are functional. In a queuing network, the queues are storage and the servers are functional. In hybrid model theory, the concept and behavior of a Petri net or a Queuing network remains identical to traditional definitions; nonetheless, in order to use these modeling formalisms in a theory which incorporates signals, the theoretical definition is more complex than it would need to be in a <u>homogenous</u> modeling theory. In symbolic analysis this is not a concern; however, in a simulation, when the signals are not part of the model, the computation should be optimized.

In hybrid model theory, a Petri net can be modeled very much like a queuing network. Therefore, only a full definition of queuing networks is developed. By describing the differences between the operation of the component models, a description of Petri nets is derived from the queuing network definition.

The two atomic models, queue and server, are developed first; then, the selective controller is defined. An atomic model which represent a queue is a storage model. The input and output of a queue represent the placing of an entity in the queue and the retrieval of an entity from a queue. The arcs in a queuing network are data flow. All storage models are <u>passive</u> in hybrid model theory. A selective controller does not operate on them. A storage component (queue) is only operated on by a functional model (a server). The queue atomic model for a FIFO queue is defined as

H :Component	<self>
A : Edge	Ø
X : Input	<type=<†,†, N>>
Ψ : Output	<entity=<†,†,Z>>
Θ : State	<count>
τ : Time Domain	<(T†,+,≤), zero, delta, map[], magnitude[]>
β : Initialize function	β[τ,χ,θ] = <k> k is an integer
δ : Transition function	Tr[τ,χ,θ]
λ : Output Function	Out[τ,χ,θ].

The initialization function initializes the number of entities in the queue. This is represented by an integer count. The transition function Tr[] changes the queue depending on the input. The input to the queue is a binary signal based on triggers. When the type signal is positive an entity is added to the list. When the signal is negative the output signal generated goes positive (if count > 0). This signifies that an entity is being transferred to a server. In this sense, the queue model operates very much like a digital counter or like a discrete integrator.

A server model is equated with a digital delay unit. When all input entities are obtained, the server delays a random amount of time before transferring entities to queues. A server atomic model can be defined as

H :Component	<self>
A : Edge	<>
X : Input	<$\chi 1, \chi 2, ...$>
Ψ : Output	<$\psi 1, \psi 2, ...$>
Θ : State	<time, state, <<que$_1$, n$_1$>,..>, <<que$_2$,n$_2$>, ...>>
τ : Time Domain	<(T,+,£), zero, delta, map(), magnitude()>
β : Initialize function	β[τ,χ,θ] = <0.0, idle, <<que$_1$, n$_1$>,..>, <<que$_2$,n$_2$>, ...>>
δ : Transition function	Tr[τ,χ,θ].
λ : Output Function	Out[τ,χ,θ].

The state set of the server model has two main entries: time and state. The state can either be idle or busy. When the state is busy, the time signifies when the server is to become idle again. The $<que_i, n_i>$ entry signifies that the server requires n_i entities from queue que_i in order to change from the idle to the busy state. If the queue is an output queue, then n_i stipulates the number of entities to put in the output queue. The transition function Tr[] for a server is defined as

$Tr[\tau,\chi,\theta] = RTr[currentTime, \tau, \chi,\theta]$ where

$RTr[t1,\tau,\chi,\theta] =$

    if $(map[t1] <= map[\tau])$ RTr[t+dt, τ, χ, state[t1,χ,θ]]

    else $<\theta>$

where

state $[\tau,\chi,\theta] =$

    if $(\theta.state = idle)$

        then if (inputAvail[θ])) $<calcDistTime[] + \tau$, busy, $<<que_1, n_1>,..>, <<que_2,n_2>, ...>>$

    else if $(\theta.time > \tau)$ putQueues[θ]; <0.0, idle, $<<que_1, n_1>,..>, <<que_2,n_2>, ...>>$.

If the state of a server is idle and the inputs are available, then the server becomes busy and new a service time is computed (calcDistTime[]). The inputAvail[] function tests all the input queues for the server. If the server is busy and the current service time is greater than current time, then the server is made idle and the function putQueues[] places entities in the output queues. Although the output function Out[] of a server can be any computable function, in hybrid model theory it is assumed to be a constant.

The selective controller model essentially has one operation. It manages when a server is allowed to obtain entities from storage. This decision must be made nondeterministically. A selective controller can be defined as

H :Component	$<self, \eta_1, ...>$
A : Edge	$<\alpha_1, \alpha_2, ...>$
X : Input	$<\chi_1, \chi_2, ...>$
Ψ : Output	$<\psi_1, \psi_2, ...>$
Θ : State	$<list>$
τ : Time Domain	$<(T,+,\leq),$ zero, delta, map[], magnitude[]$>$
β : Initialize function	$\beta[\tau,\chi,\theta] = <list>$
δ : Transition function	$Tr[\tau,\chi,\theta]$
λ : Output Function	$Out[\tau,\chi,\theta].$

The state of the selective controller is an unordered list which contains the functional models (servers) of the component set C. The initialization function produces the initial list. As with state controllers and parallel controllers, the output function and the transition function of a selective controller are very similar. The transition function $Tr[]$ is defined as

$Tr[\tau,\chi,\theta] = RTr[currentTime, \tau, \chi,\theta]$ where

$RTr[t1,\tau,\chi,\theta] =$

    if $(map[t1] <= map[\tau])$ $RTr[t1,\tau,\chi,<state[t1,\tau,\chi,\theta]>]$

    else $<\theta>$

where

$state[t1,\tau,\chi,\theta] =$ For all $s_i \in \theta.list$ active$((s_i.Tr[t1, t1+deltaT, X, s_i.\theta))$

The transition function again consists of one recursive function and one computational function. The recursive function increments the time. The computational function $state[]$ nondeterministically invokes all the functional models (servers) one at a time . The general approach to this definition is exactly the same as how a queuing network or Petri net would nondeterministically choose servers or transitions. The event is stored locally by each server or

transition (the time variable in the server's state Θ). At each time quantum in a simulation, the servers are checked to find out if their next event time has occurred.

The output function of a selective controller model is a sequence of invocations of the functional models output functions. Therefore, only one functional model (server) can supply an output. This does not preclude one server from suppling more than one output. It should be noted that the time domains of a selective controller and its servers must be the same. Otherwise, there would be errors in allocation of entities among servers.
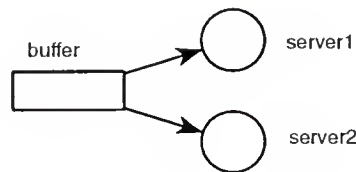

Figure 4.6 Queuing Network

An illustration of the selective controller is given by identifying the operational definition of the queuing network in Figure 4.6. There is one input and one output signal. This definition is

```
QueueNetwork[startTime, endTime, input]  {
    initialize[];
    tempTime = startTime;
    while (tempTime < endTime)
      {parbegin
          {if (active(server1)) server1.Tr[tempTime, tempTime + deltaT, input]
          if (active(server2)) server2.Tr[tempTime, tempTime + deltaT, input]
          }
      output =  server1.Out[tempTime, tempTime + deltaT, input]
      stillActive(server1)
      stillActive(server2)
      tempTime = tempTime + deltaT
      };
    return output;
}.
```

The operational definition also makes the intramodel coordination clear. When the output function of a server is invoked, if the server was submodeled, then the submodel would be invoked instead of the server model as defined above. For instance, if the server had a block diagram submodel, then the server's output would be computed from the block diagram. This only requires that the block diagram submodel have the same input and output specifications as a server model.

A Petri net is a specialized queuing network where the queues are places and the servers are transitions. The inputAvail[θ] function is based on an "and" conditional instead of an "or" conditional. The transitions are typically modeled with a constant firing time (called the service time in queuing networks). The places of a Petri net act like unordered queues. If the time domain of a Petri net is TD_3 and the transitions have a firing time of 1 unit, then the behavior of a Petri net in hybrid model theory has the same behavior as the Petri nets as defined in Chapter 3; however, the elapsed time in hybrid model theory for selective controllers does not signify the number of transitions fired.

CHAPTER 5
HYBRID ANALYSIS

KAS Modeling

The following discussion is based on the process an investigator might go through in an attempt to model an automated flexible manufacturing system (AFMS). This scenario will demonstrate how an investigator can select formalisms to suit the pragmatic issues at hand. The three tasks outlined in Figure 2.1 of Chapter 2 are demonstrated with this example.

An AFMS was chosen as the domain because of the complexity involved in modeling the many aspects of such a system. Primarily, an AFMS is a system consisting of several work cells and a transit system. A computer system acts as the controller for the entire system (i.e., scheduling operations, resolving conflicts). Each work cell is a logical unit consisting of a set of machines or robots. Within a work cell, many different types of operations may occur, for example, milling, drilling, pressing or assembly. Machines and robots operate on parts (these may simply be raw materials). Parts which will be assembled together or require similar operations are said to be in the same family. The general sequence of events in the AFMS entails transporting parts to a work cell, performing specific operations and transporting parts back to a storage area. The complexity is associated with the control of this system. This involves scheduling operations, allocating machine and transport resources, buffering intermediate parts and final assemblies and decisions as to when parts are to be manufactured to meet current or projected demands.

As a specific example, Figure 5.1 shows a typical layout of an AFMS. There are two main work cells in this example: the machining work cell and the assembly work cell. The arrows

indicate the path of the transport system. The machining work cell has several machines which prepare parts for assembly or further machining (although it is possible to machine parts as part of an assembly). The assembly work cell consists of several robots which put the parts together. In this example, the main transport system is an automatic guided vehicle system (AGV). The AGVs retrieve and store pallets. The pallets are removed from storage at a loading dock and put onto an AGV. The AGV transports the pallets to unloading docks at the work cell and transports finished parts back to storage or other docks.
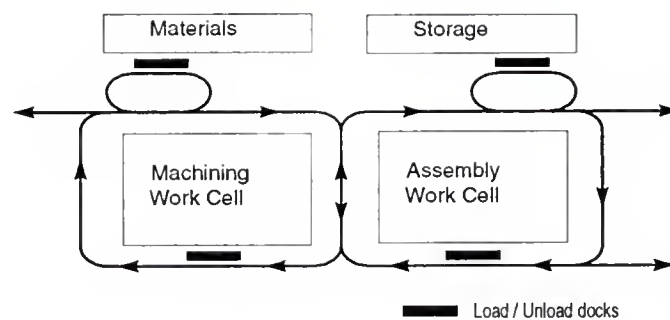

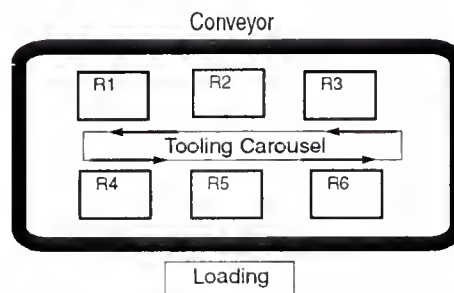
Figure 5.1 - Example AFMS Floor Plan

Figure 5.2 - Machining Work Cell Floor Plan

Figure 5.2 shows the machining work cell in more detail. There are 6 machines in this work cell (R1-R6). The work cell has its own transport system (a conveyor). The conveyor transports pallets around the work cell so that a machine can remove a part, work on it and put the part back on a pallet. When a pallet has all its parts completed, it can then be picked up by an AGV or stored in a buffer at the loading dock for later pick up.

Each machine can perform any number of operations. All that is required is that the appropriate tool be available. The tooling carousal continually transports tools around the work cell so that when a machine needs a tool, the machine can remove it from the carousal. Scheduling and resource allocation play a major part in the efficiency of the work cell.

The assembly work cell is similar to the machining work cell except that an assembly cell has robots instead of machines (robots being slightly more dextrous than machines). Figure 5.3 is an example of the layout of a robot's work area. When a robot is scheduled to assemble parts, the parts must either be in the robot's buffer, on the robot's assembly platform or on a pallet on the conveyor. Once an assembly is made, it can be transferred to a pallet or the buffer.
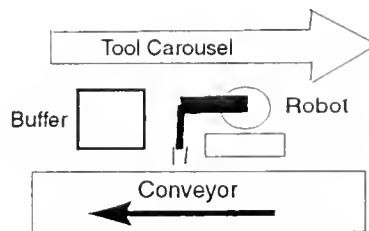
Figure 5.3 - Robot Cell Floor Plan

Figure 5.4 - Partial Concept Model of AFMS
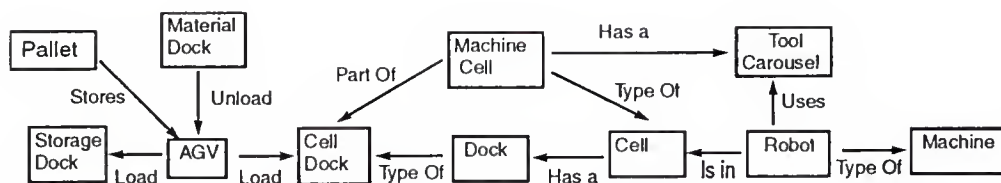
Figure 5.4 shows how a semantic model of part of the real system would be organized. This is essentially a concept model in SE [Rum91] or a semantic net in AI [Fin79]. The automatic conversion of this model to an executable model is not part of this research; however, once a dynamic model is built, there are questions that can be answered which are not obtainable by the static model.

Heterogeneous Hierarchical Modeling

The current emphasis in industry today on total quality management (TQM) and design for manufacturing has reemphasized the need to model very large diverse systems. Instead of modeling a factory floor, for instance, the emphasis on TQM requires the model of a AFMS to include economics, consumer demand, product distribution, etc. With this in mind, the context of an AFMS similar to the above example is modeled in this section. Figure 5.5 demonstrates a simple, initial model of the highest level of abstraction. As this modeled is refined, it will become clear why this model is abstract. There are three main interacting subsystems: a producer, a consumer and a pricer. The initial goal of the investigator will be to minimize the size of the storage while still meeting consumer demand. Although not explicitly shown, the time domain for this model has significant measures of time in terms of days. The pricer function includes modeling marketing policies decisions such as trade promotions. The consumer function includes modeling behavior such as brand loyalty. The main emphasis in this example is on the producer function.

The producer function uses the current size of the storage as the only influence on the production rate (dP/dt). Because consumer demand (dD/dt) can fluctuate, the factory usually buffers a certain number of units of the product. The allowable size of the storage buffer has a minimum and a maximum size (Min, Max). The time domain for this model also has significant measures of time in terms of days. In order make maximum use of resources, there will be three different states of the producer function (factory) depending on the state of the storage. In Figure 5.6, the factory is modeled as a state machine. The three states indicate whether current production is below, above or within the specified buffer limits. The two names producer and factory actually refer to the same functional model and can usually be used interchangeably. producer is the name of component model and factory is the name of the submodel. This allows the investigator to name a functional model depending on the context in which it is referenced.
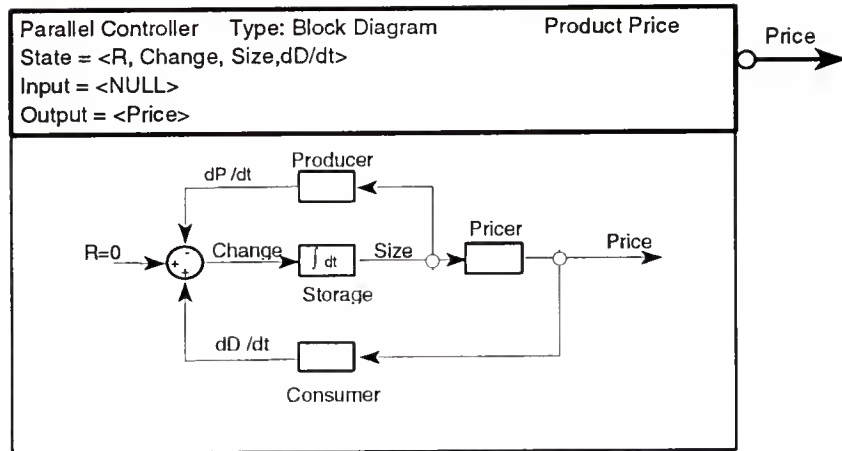
Figure 5.5 Block Diagram of a Product Price

Figure 5.6 State Machine of Factory (Producer)

In each state of the factory, the number of machines and people allocated may or may not

vary (analysis will determine this). In the shutdown state, the production rate (dP/dt) is assigned

zero (output function at any time t equals 0.0, Shutdown.λ[t] = <0.0>). The production rate in the

overtime state is left undefined (OverTime.λ[t] = †). In the Normal state, the production rate

equals the rate of demand set by initial marketing studies. It is decided that two robot assemblers

are needed to meet this demand.

An important influence on the robot's ability to meet this demand is the down time of each robot; the normal state, therefore, is a combination of four states: both robots working, both robots down and 2 states with one robot down. An effective way to model this situation is with a Markov system. Figure 5.7 depicts the down time model. Notice that the input and output of the Markov controller model include the input and output of the state machine. This is required by hybrid model theory. Although the input to the Markov system is not used by the component models, it can be "passed down" to submodels of the individual Markov states. The time domain for this model has significant measures of time in terms of hours.
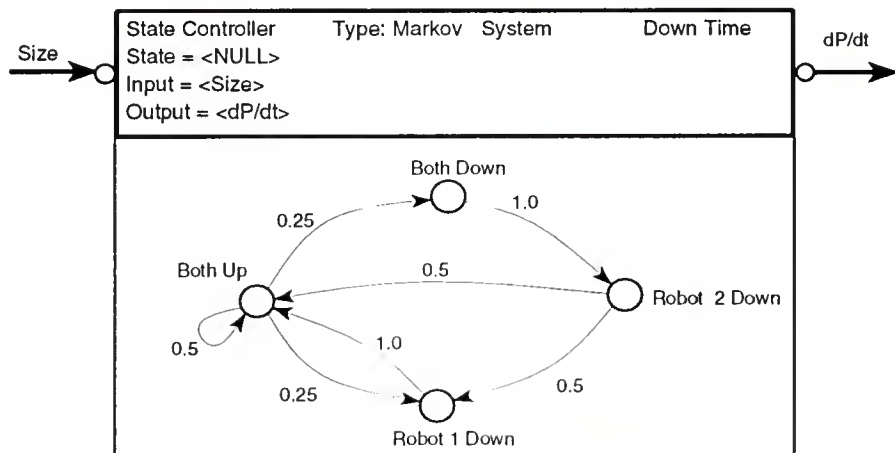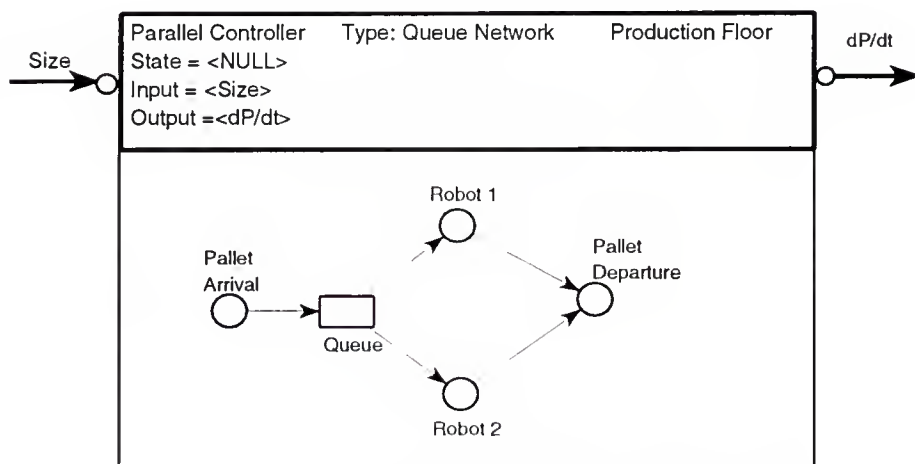

Figure 5.7 Down Time Markov Model of Robots

Figure 5.8 Production Floor Queuing Network

The models of the both up, Robot2 down, and Robot1 down state can be effectively modeled with queuing networks. This is exemplified by showing the both up state in Figure 5.8. This is a simple two server, first-come-first serve queuing network (specifically a M/M/2/ ∞ /FIFO/system). Again notice the input and output to the queuing network controller model. It must include the input and output of the Markov state supermodel even though queuing networks do not have external data input and output. The time domain for this model has significant measures of time in terms of minutes.

It is important to understand that the input and output do not alter the production floor queuing network. The output (dP/dt) can only be derived from a property of the queuing network. The input (Size) can be used to derive the output (dP/dt) or passed down to submodels. The derivation of the production rate does not interfere with the behavior of the queuing network. The same idea holds for all the models which have been already been constructed. The derivations are coordinated with the behavior: they do not replace any behavior or alter any behavior of the formalism being modeled.

It is assumed that the current model description is sufficient to describe the production rate (dP/dt); therefore, in order to have a complete model, the output of the queuing network must be the production rate. The production rate in this case is the departure rate of the pallets. Figure 5.9 shows the departure component model and how the output is supplied. The other component models in the queuing network cannot supply output to the production rate (Chapter 4).

Pallet Departure   Type: Atomic (queue)
State = <count>
Input = <input>
Output = <dP/dt>
$\lambda[\tau, \chi, \theta] = $ (X.input[τ - deltaT] - X.input[τ]/ deltaT)
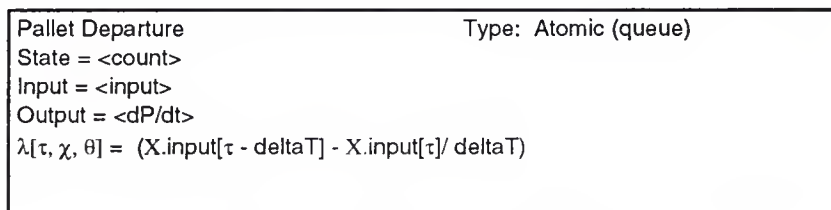
Figure 5.9 Departure Model of pallet

If the investigator did not supply the connection between the pallet departure node and the production rate (dP/dt) of the controller model, then numerical analysis could not be performed. However, in symbolic and interpretation analysis, the model could still be effectively used by inventing a symbol for the production rate.

Hybrid Analysis

The benefits of hybrid model theory can now be demonstrated by taking advantage of the clear semantics and symbolic methods of the formalisms used. The process of modeling the AFMS in the last section is not just a graphical interface method for a simulation language (like TESS [Sta87] is for GPSS). Although graphical interfaces provide significant increases in efficiently, it is the implicit semantics of the formalisms which when combined with hybrid model theory, allow the whole to be greater than the sum of its parts. An HH model is really a single representation which allows for all three types of analysis.

The numerical analysis (model simulation) of a hierarchical model is very straight forward. Each of the formalisms which was used has a well defined numerical method. With hybrid model theory, the input and output relationship between models and the manner in which each model handles time has been formally defined. Intramodel coordination is accomplished by formally describing a model with two levels: controller and components. The transition and output functions specify the computation to be performed at each step for each model. The time period is supplied as an input to the simulation. Each model's delta time is examined to determine the minimum time slice required for the simulation. Those models whose delta times are greater than the simulations time period are considered to be static systems. The operational definitions in Chapter 4 are, for all practical purposes, a compilation of hybrid models into numerical analysis models.

To perform a numerical analysis, the entire model must be checked for completeness (compiled). The compiler requires default values for variables at the leaf models in the

hierarchical model, but does not require a complete conceptual model (by numerical values fuzzy numbers are also valid [Zad75]). It would be convenient if the system could set up default initial conditions for the investigator; however, this is not the purpose of HH modeling or hybrid model theory. The importance and benefits of automatically setting up the simulation is not deemphasized, but this research is limited to efficiently creating heterogeneous hierarchical models and supporting hybrid analysis techniques. If this can be accomplished, the ability to integrate information from symbolic, numerical, and interpretation sources will promote the automation of the more complex tasks such as setting up simulation runs and approximating submodels.

Symbolic Analysis

Many symbolic analysis routines are specific to individual formalisms; there are, however, some simple general routines which might be useful in very large systems. The singularity of these routines does not prevent a categorization of symbolic routines. For example, if steady state analysis of a model is required, any formalism which can potentially obtain steady state information must define a steadyState[] symbolic analysis function. Formalisms such as block diagrams, Markov systems, and queuing networks could derive the traditional attributes associated with the notion of steady state. Formalisms such as state machines and Petri nets have no traditional notion of steady state. In the context of hybrid symbolic analysis, the routine steadyState[] for state machines and Petri nets must still be provided, but, at the very least, they only need to formulate a symbolic value to represent the steady state. The need for this will be demonstrated in the next section.

In the AFMS model presented in the last section, one of few numerical results from symbolic analysis which can be obtained by a single formalism is the steady state of the down time model. Because the down time model is a regular Markov system, the steady state equilibrium probabilities of each state can be found. This is

$Lim_{n \to \infty} (M^n)i,j = SS(s_i)$

where $SS(s_i)$ is the Steady State probability for each state i

$M_{i,j}$ is a matrix representing the probability of going from state i to state j.

The steady state SS(si) can be represented by the creation of a vector within the system (downTimeProbabilities = [0.133, 0.2, 0.533, 0.133]). Here, the first element is the steady state probability of being in the both up state, the second element is the steady state probability of being in the Robot1 down state, etc.

A purely symbolic result for several attributes of interest can also be obtained from the queuing network. Because the network was defined as a M/M/2/∞ /FIFO system, the expected number of pallets in the model is as follows [Gra80].

$$L_{palette} = \frac{\lambda}{\mu} + \left[\frac{(\lambda / \mu)^C \lambda\mu}{(C-1)!(C\mu - \lambda)^2} \right] P_0$$

$$P_0 = \left[\sum_{j}^{C-1} \frac{1}{j!} \left(\frac{\lambda}{\mu}\right)^j + \frac{1}{C!} \left(\frac{\lambda}{\mu}\right)^C \left(\frac{C\mu}{C\mu - \lambda}\right) \right]^{-1}$$

where C = 2, the number of servers

λ = the arrival rate

μ = the service rate

The most interesting results of symbolic analysis result when a model is required to obtain information that is not part of its semantics. For example, suppose the expected number of pallets in the normal state of the factory model of Figure 5.6 is needed (i.e., What is L_{normal} ?). The normal state model can not find this information with symbolic methods; however, because there is a submodel, the normal state symbolic routine calls the submodel symbolic routine to find

the expected number of pallets. The submodel is a Markov system. It also can not determine the expected number of pallets with symbolic methods. If its components submodels could find this information, then the result of the Markov model would be the steady state vector SS (down time probabilities) times a vector of the individual expected number of pallets for each of the four states s_i (the dot product). This is

$$L_{normal} = \text{down time probabilities} * L_{pallet}[],$$

$$L_{pallet}[] = [s_1, s_2, s_3, s_4]$$

or

$$L_{normal} = \text{prob[both up] } (s_1) + \text{prob[Robot2 down]}(s_2) + \text{prob[Robot1 down] } (s_3) +$$
$$\text{prob[both down]}(s_4).$$

The individual Markov state symbolic routines are then called to derive the expected number of pallets. The three state that were not submodeled (Robot2 down, Robot1 down, both down) also cannot find the expected number of pallets; therefore, they return a symbolic result. The queuing network submodel can return a result as previously shown. This symbolic result is returned to the Markov system model of the factory. The Markov system then returns the result to the normal state. The final result would be

$$L_{normal} = 0.133 \, (L_{pallet_0}) + 0.2 \, (L_{pallet_1}) + 0.533 \, (L_{pallet_2}) + 0.133 \, (L_{pallet_3})$$
where $L_{pallet_i} = \text{unknown, for } i = 1\text{-}3$ and

$$L_{pallet_0} =$$

$$\frac{\lambda}{\mu} + \left[\frac{(\lambda/\mu)^2 \, \lambda\mu}{(2\mu - \lambda)^2} \right] P_0$$

$$P_0 = \left[\left(\frac{\lambda}{\mu}\right) + \left(\frac{\lambda}{\mu}\right) \left(\frac{2\mu}{2\mu - \lambda}\right) \right]^{-1}$$

where λ = the arrival rate

μ = the service rate.

The necessity for all models to handle routines like steadyState[] and expectedNumberOf[] is demonstrated by this last example. Even thought a routine can not find a property, it must be able to construct a result if the component submodels exist or it must return a symbolic result for its supermodel to use in the construction of a result. This recursive technique is based on intramodel coordination (submodeling). Additionally, the combination of the type of data needed (continuous time, value) and model state (steady state, single point in time, specific time period, specific condition) dictates how a modeling formalism must respond. For instance, the expected number of pallets is a steady state attribute; therefore, models must respond accordingly. On the other hand, if the production rate (dP/dt) when (Size > Min) was of interest, then each model would be required to construct the result for a continuous time, discrete value variable under the specific conditions (Size > Min). The question would be recursively asked of each submodel. Those controller models which could not construct a result either because the component models were not submodeled or because the question was not appropriate would return a symbolic result.

For questions which resulted in unsatisfactory answers (too many "invented" symbols), numerical analysis would be a possible next step. For example, in the both down state of the Markov system, the production rate should be zero since neither robot is working. It is not expected of the symbolic routines to find this answer. A numerical analysis would implicitly find this result. With a more complete model, the interpretation (next section) could deduce such information from knowledge that the production rate (dP/dt) and pallets are numerically related (Figure 5.9).

The results of the symbolic analysis could be used to help set up the boundaries under which the numerical analysis operated. For example, in the above analysis of L_{normal}, the simulation of the model to find a refined answer (a specific distribution) would not require a simulation of the queuing network system. This could be replaced by an appropriate distribution determined by the result of the both up state (L_{pallet}). Thus, it would reduce the time necessary to obtain the computational results.

Interpretation

Correct interpretation depends on using a consistent set of labels on the graphs of the models. At first, this may seem too restrictive; however, if one realizes that the model is a fact base, then the model is really a special derivative of a semantic net that describes the dynamics of the system under investigation. Each formalism has several consistent naming schemes which can be compiled into a consistent fact base. The knowledge base is a set of rules which are derived from controller's transition function and the semantics of each individual formalisms. These are called formalism rules. For instance, qualitative reasoning methods [Bob86] form part of the knowledge base for block diagrams.

Together with the fact base, the knowledge base can interpret the model. A pseudo Prolog representation is used to present the interpretation process. It is assumed that the interpretation process uses a goal directed deduction algorithm (also one which allows for truth maintenance). However, other knowledge representations are certainly feasible. It is also assumed that a natural language parser (NLP) converts questions into the pseudo Prolog format. With the availability of an online dictionary and thesaurus which can find plurals from single nouns, identify parts of speech, check grammar, and provide synonyms, such a system is certainly attainable (also, with the availability of parser support such as LEX and YACC). The NLP will be far from perfect, but it would release the investigator from being required to understand the syntax of Prolog. Additionally, it is an intricate part of a domain independent, generic knowledge base. This issue is discussed as the interpretation process is presented.

One of the simplest types of formalism semantics to demonstrate is that of a state controller. There are two consistent labelings:

1. All nodes are labeled with noun phrases (Figure 5.6).

2. All nodes are labeled with verb phrases.

Along with the node labels, the type of phrase which describes the submodeled component and controller model is needed. The factory submodel of the producer model (Figures 5.5 and 5.6), has a sequence of noun labels (normal, overtime, shutdown) describing a noun phrase (producer). Here, the noun labels are interpreted as the states of the producer. These interpretations depend only on the types of phrases used (only 2 in the current research: noun and verb) and the type of controller used (only four). A cross product of these produces a finite set of interpretations for state model nodes. From the models in Figures 5.5 and 5.6, a computer environment could automatically build the following simple facts:

stateOf[producer, normal],

stateOf[producer, shutdown] and

stateOf[producer, overtime].

It is assumed that the predicate stateOf is used in the generic knowledge base rules. For example, the following rule attempts to find the conditions necessary for C (if any) to go from one state X to another state Y by using the stateOf predicate:

nextState[M, X, Y, C]:- stateOf [M, X], stateOf[M, Y], trace[M, X, Y, C].

As can be seen form the first rule, if both states are in the same model, the trace[] predicate is called with the model and the two states as input and expected to determine, depending on the type of model, the conditions necessary to go from state X to state Y. A simple test for the trace could be to call a traceState[] predicate which follows all paths in a state model:

traceState[M, X, Y, (C1C2)] :- type[M, State], nextState[M, X, Z, C1], notVisited[X, Z],

traceState[M, (XZ), Y, C2] and

traceState[M, X, Y, C] :- NextState[X, Y, C].

The notVisited[] predicate avoids infinite loops and "()" represents concatenation. If this failed, the trace[] function can use methods such and constraint propagation and qualitative simulation to derive an answer (if possible) for the nextState[]. Note that the NextState[] facts can be derived directly from knowledge about formalisms (the hybrid model). From Figures 5.6 and 5.7, the following can be derived:

NextState[overtime, normal , (Size<Min)] and
NextState[both up, Robot 1 down, Stochastic (0.25)].

These predicates are very primitive, but they can be used to build more sophisticated predicates. The nextState[] and trace[] predicates can be used to build an activeStates[M, C] predicate which finds all the states of model M that could be active under the conditions C. For instance, as can be seen in Figure 5.6 the function invocation

activeState[factory, (Size < Max), state?]
returns
state = normal, state = shutdown.

That is, when the Size < Max in the factory model, both the normal and shutdown states models could be active.

How each type of controller uses the notion of state and how states are relate between supermodel and submodel would obviously need more rules than listed here. Again, it is emphasized that the goal is to efficiently build models and allow a set of formalism rules to operate on the model with only the minimum possible effort of the investigator. No attempt is being made to present a concise set of formalism rules. The objective of this section is to show that the implicit semantics of the formalism and hybrid model theory, applied in a hierarchical fashion, allow for quick and efficient model development by the investigator and automatically provide fact bases used in interpretation.

Another illustration of interpretation can be demonstrated with consistent labels used on the queuing network presented in Figure 5.8. Some of the rules for labeling are as follows:

1. arrival and departure nodes use the same noun phrase and

2. servers are either all nouns or all verb phrases.

Figure 5.8 shows that pallets are the entities which arrive and depart in the queuing network. If parts were the entities that arrive and assemblies were the entities that departed (certainly a feasible arrangement), then static information would be required which related the two (possibly in the form of a semantic network). The static information would stipulate, for instance, that an assembly is made of parts (madeOf[assembly, parts]). Combining semantic nets and mathematical formalism like queuing networks in the context of HH modeling is discussed in Miller [Mill92].

Simple state information can be derived from the semantics of the queues and servers labeled with nouns. A few facts from Figure 5.8 are listed here:

state[M, empty] :- Not (stateOf[pallet Queue, in-use]),

state[M, idle] : - Not (stateOf[Robot 1, busy]),

stateOf[M, empty] :- type[M, Queue], state[M,empty],

stateOf[M, in-use] :- type[M, Queue], state[M,in-use] ,

type[pallet queue, Queue] and

traceState[M, X, Y, C]: - type[M, Queue], (X = empty), (Y = in use), (C = arrives[M, data]).

Even though the the queuing network is a selective model and not a state model, State information can still be deduced. For instance, the question

nextState[pallet Queue, empty, in use, C]

is easily traced to

C = arrives[pallet Queue, pallet].

The generic use of terms like busy, waiting, empty, and in-use in the fact base does limit the domain of the generic knowledge base. For instance, an investigator might ask "Is Robot1 active?". It would appear that the internal form would be stateOf[Robot1, active]. However, the NLP, through the use of a online thesaurus/dictionary, can easily translate this into stateOf[Robot1, busy]. All that is required is that the NLP have a list of words that the interpretation system can use. The NLP simply translates into the appropriate words. This frees the interpretation system and its rules from being domain specific. For example, in a bank queuing network the question might be "Is the teller working?". Because the words active and working are both synonyms for the word busy, the translation is trivial.

The use of an NLP to do this translation is not an afterthought in hybrid model theory and is extremely important for knowledge-based access to dynamic models. Hybrid model theory was developed around the concept of "as much domain independence as possible with as many analytical methods available." It was found that formalisms like Petri nets had well-defined dynamic semantics, but had no easy way to relate the semantics to specific domains. Knowledge-base techniques tended to be domain specific, but had no easy way to build up the complex dynamic operations of the systems to be modeled. By using a few simple rules to ensure consistent labels, using named structures as an operational part of hybrid model theory (not just a convenience) and taking advantage of the online thesaurus/dictionaries, an investigator can quickly construct a model that had well defined formal properties and can be used by a domain independent knowledge base.

The thesaurus/dictionary acts as the semantic knowledge base which translates questions between the specific domains and the generic knowledge base. The domain objects (pallet, robot) in the knowledge base are derived from the named sets of the model. The relationships between the objects (is working, is busy) are derived from the semantics of the formalisms and hybrid

model theory (intramodel coordination in particular). The names of the relationships between domain objects are derived by using the semantics of the formalisms and by the use of consistent labeling. The fact base is a set of primitive facts and rules compiled from the model. When combined with the generic knowledge base, a very large domain specific knowledge base can automatically and quickly be created (compiled) from an HH model.

To further demonstrate how interpretation can be used, the trace of a more difficult question is presented. The question is " What is the production rate (dP/dt) in the normal state of the factory". The following internal forms can automatically be generated:

dP/dt == "production rate",

modelcontext == normal and

goal[dP/dt, modelContext].

Here, it is assumed that questions are ultimately put in the form of a goal. The predicate for this is goal [variable*, context], where variable* means any number of variables and the context is a description of the system current state. The information that the normal state is in the factory model is useless in the current model because only one state in the entire system is labeled as "normal."

One of the first conclusions an interpretation environment can easily make is that of the time period over which the context holds. This is one of the uses of the model's time domain. It can be concluded that the factory model is constant for the time in the question since a single state of the factory is of interest and the magnitude function shows a significant difference in time (normal.τ.magnitude[hours] $\equiv$ factory.τ.magnitude[hours] $<<$ price.τ.magnitude[days]).

It can now be shown how the interpreter attempts to formulate the answer to the question. Once the goal predicate is derived, the interpreter can direct the question to the modelContext. In this example, the knowledge base for the normal model (a state) is asked to find dP/dt. This is represented as stateFind[variables, Model].

If the normal state model was not submodeled the interpreter could return the value set up by the investigator. If there was no specified value, then "undefined" (†) could be returned. However, since the normal state model is submodeled, the interpreter (via a rule) redirects the question to the Markov system submodel. This is represented as markovFind[variables, Model].

In the Markov system in Figure 5.5, the Robot1 down, Robot2 down, and both down states did not relate the production rate dP/dt to the output of the model, the interpreter has to determine whether there is or is not output from these states. The requirement that all data be specified as continuous(discrete) over value and time now comes into play. Because the production rate (dP/dt) is continuous time and discrete value and the controller is a state controller, the interpreter can deduce that all states output to the variable dP/dt. Therefore, it can "invent" symbolic names for the unspecified values from these states. For example, in the Robot1 down state, the interpreter would construct the symbol "Robot1 down dP/dt." Recall, that for numerical analysis, these missing values would have to be supplied in order for compilation to be completed.

The interpreter, currently within the markovFind[] rule set, can derive the following list of assignments:

state dP/dt = [pallet departure, Robot1 down dP/dt, Robot2 down dP/dt, both down dP/dt],
down time probabilities = [0.133, 0.2, 0.533, 0.133] and
pallet departure = down time probabilities * state dP/dt.

The state dP/dt variable is a temporary set up by the interpreter. The last three entries (Robot1 down dP/dt, Robot2 down dP/dt, both down dP/dt) in the state dP/dt vector were derived by the interpreter as stipulated in the last paragraph. The pallet departure entry was derived from the queuing network model. The interpreter redirected the find predicate form the both up state model to the production floor queuing network submodel via a predicate such as quenetFind[variable, Model]. This recursive technique is very similar to the recursive technique

used in symbolic analysis and is made possible by intramodel coordination. Because the queuing network model is a parallel controller, the interpreter knows that there is only one collective state. If the pallet departure model was not connected to the output dP/dt of the queuing network, then the interpreter would have "invented" a symbolic value; however, because of the connection, the interpreter knows that the pallet departure is the only source for the production rate dP/dt.

The "pallet departure = down time probabilities * state dP/dt" expression is derived from assumptions about Markov systems. The "down time probabilities" is symbolically derived at as shown in the last section. Notice, that the assumptions made by the interpreter are really rules which apply to controllers. The rules presented so far as can be stated as follows:

1. if (output type = continuous time, discrete value) and (controller = state)

   then (all states must supply output),

2. if (output type = continuous time, discrete value) and (controller = parallel)

   then (find submodel with output) or (invent a name for output)   and

3. if (output type = continuous time) and (model is Markov system)

   then (output = findProbabilityVector()* vector [individual state outputs]).

These rules demonstrate how hybrid model theory allows knowledge about different formalisms to be generalized. If a new formalism which uses a parallel controller were added, then rule 2 above would still apply. It also demonstrates how one analytical form can help another. Here, symbolic analysis was used to aid in the interpretation (the findProbabilityVector() function invocation).

CHAPTER 6
CONCLUSIONS AND SUMMARY

Conclusions

There are several problems which may or may not reduce the effectiveness of hybrid model theory. Although other formalism can be used, only five formalisms have been explored. Some formalisms theoretically fit within hybrid model theory, but they do not fit not conceptually. For instance, in back-propagation neural networks, nodes (neurons) traditionally do not have meaning. Only a few types of neural networks, such as those which use policies like harmony theory, have meaningful nodes; therefore, labeling a back-propagation neural network has no significance. However, submodeling a neuron may have significance.

In general, labeling nodes requires some extra effort. It is certainly much easier than starting a knowledge base from scratch: It is also better than having no knowledge base at all. The fact base will only be sufficient for complex questions if enough detail has been modeled. Additionally, because the fact-base is developed strictly from formalisms which are based on time, only dynamic information can be derived. Some simple typeOf and partOf information can be found from the hierarchy. For example, in the AFMS the Robot1 was part of the factory (partOf[factory, Robot 1]). This is not enough information to ask questions about physical structure or the geometry of the system (i.e., Will any of the AGVs collide?). In KAS models, the physical location of objects will change over time. It is reasonable to expect that in some systems the physical relationships between objects over time is the information which is sought by the investigator. In hybrid model theory, there is no clear means to coordinate geometric formalisms.

Another problem is the relationship between different symbolic analytical methods. Each routine works individually. In some cases, it is apparent how one method can help another; for

example, a symbolic method can help a numeric method. In general, the knowledge base will also have to include information pertaining to the relationship between properties derived from the analytical methods. This was expected, but the approach can be highly individualized. Adopting a particular methodology may not be desirable. Committing to a method which stipulates a specific sequence of the principles may not have been the most appropriate method, but it may be unavoidable if multiple sources of information are to be utilized.

Another problem is the selection of the initial model. Because the main emphasis may be on systems about which very little was known, the initial choice of a model could not easily be determined (e.g., state versus parallel). Therefore, it might be necessary to include an abstract formalism in hybrid model theory. This would let the investigator begin to describe objects, relationships, and data without committing to a specific approach. When the investigator does commit to an initial approach, the abstract formalism (really just a hierarchical graph) serves only to group together the underlying models. The abstract model has no dynamic description or useful analytical properties; However, the graph can be used to compile simple facts for the knowledge base.

In selective controller formalisms, the tokens (Petri nets) or customers (queuing network) might represent objects which an investigator would like to model. There is currently no clean way to describe formally the interaction between these objects and the selective controller model. The investigator must program the interaction by hand. One of the goals of hybrid model theory is to remove this type of ad-hoc or informal coordination.

A hybrid model represents a structural or conceptual hierarchy. Class hierarchies do not conform to the paradigm enforced by hybrid model theory. In some cases, especially when building libraries of models, the structural approach is not suitable; for example, when a typeOf hierarchy is needed. Hybrid model theory only permits partOf hierarchies.

With hybrid model theory, there is at least a formal foundation upon which these problems can be formulated. Without a standard theory, there is no way to analyze, symbolically or numerically, a model in a general manner. Without a standard theory, in either intra or

intermodel coordination, each pair of formalisms would require a distinct coordination protocol and would require formalism specific knowledge.

Hybrid model theory is based on formalisms (Petri nets, queuing nets, etc.) which are used in many fields and understood by many researchers. This places hybrid model theory in a unique position within modeling theories. It has extended the potential of formal modeling methods without changing the way investigators currently use modeling methods and without significantly increasing the complexity involved in modeling.

<u>Summary</u>

Heterogeneous Hierarchical modeling is a general term used to describe any method which supports the construction of models in a hierarchical manner with the use of multiple model types. The method must also support the use of symbolic, numeric, and interpretative analysis methods. Hybrid model theory is a theoretical representation which provides the necessary formality to meet the requirements of HH modeling. This is accomplished by coordinating existing modeling formalisms instead of trying to take a single formalism and generalize it.

In hybrid model theory, an investigator can easily construct a model by iterative refinement. Each level in the hierarchy increases the detail and accuracy of the model's behavior. Because each level in the hierarchy is more abstract than lower levels, a hierarchical model provides an information structure compatible with knowledge base reasoning methods; however, each level, even the most abstract ones, are modeled using formalisms which have well known symbolic and numeric properties. Consequently, a <u>partial</u> or <u>abstract</u> model can still be analyzed. The hierarchy also supports developmental and conceptual efficiency for the model builder by allowing for structured, top-down development.

The most effective mechanism for cultivating developmental and conceptual efficiency is achieved by coordinating heterogeneous model types. At any level in the hierarchy, an investigator can use a formalism which meets his/her current functional and pragmatic needs. In hybrid model theory, the modeling formalism which captures the essential characteristics of the

behavior currently being modeled can be coordinated with models of other parts of the system regardless of the formalisms used to describe them.

Hybrid model theory conceptually separates formalisms into two levels: controller models and component models. Three of the three types of controllers were discussed: state, parallel, and selective model controllers. These three controller types support intramodel coordination. Each of the three model controllers manages a set of component models. For example, a Petri net is a selective controller that manages place and transition component models. By creating two levels, three goals of HH modeling have been realized. First, a component can be submodeled (coordinated) with another completely different type of formalism (represented by another controller model). Second, knowledge about controllers can be generalized. This simplifies the inclusion of additional types of formalisms. Third, intramodel coordination provides a structured way for hybrid symbolic and numeric analytical methods to be used in conjunction with each other.

Hybrid analysis also includes traditional AI and knowledge base methods. By taking advantage of natural language text attached to the nodes and arcs of the model's graph, a fact base compiled from this text can be combined with a generalized knowledge base. The investigator needs only to follow a few simple consistency rules regarding the form of the text. The compiled knowledge base can then serve as a front-end in a computer environment and as a source of new information by the use of reasoning techniques.

A hybrid model is a declarative representation of a system. The component, edge, and state set contain the necessary information for symbolic analysis. The state set, transition function and output function contain the information required for numerical analysis (simulation). The use of named sets and projection functions allows the automatic compilation of a knowledge base.

In this context, hybrid model theory is a theoretical representation which provides the necessary formality to meet the requirements of heterogeneous hierarchical modeling. Hybrid model theory is an alternative approach to combined discrete-continuous multimodel theories and subsumes most of the concepts in combined discrete-continuous system simulation. It supports a

new concept in heterogeneous hierarchical modeling called intramodel coordination. Intramodel

coordination is a method in which the components of a model can be coordinated with other

models. In this manner, hybrid model theory extends the notion of intermodel coordination in

combined discrete-continuous system simulation. Intermodel coordination is a method in which

two models can only interact through input and output. Furthermore, a hybrid model is a

declarative representation of a system. By restricting the form of this representation, a hybrid

model contains the data necessary information for a computer environment to perform symbolic,

numerical and interpretative analysis automatically without additional effort from the user.

REFERENCES

All83   Allen J.F., "Maintaining Knowledge about Temporal Intervals," Communications of the ACM, 26(11), pp. 832-843, 1983.

All84   Allen J.F.,"Towards a General Theory of Action and Time," Artificial Intelligence, 23(2), pp. 123-154, 1984.   .

Ban84   Banks J., and J.S. Carson II, Discrete-Event System Simulation, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

Bir79   Birtwistle G. M., Discrete event modelling on simula, Springer-Verlag, New York, New York, 1987.

Bob86   Bobrow D.G (ed), Qualitative Reasoning about Physical Systems, MIT Press, Cambridge, Massachusetts, 1986.

Cel82   Cellier F.E. (ed), Progress in Modelling and Simulation, Academic Press, London, England, 1982.

Cel92   Cellier F.E, "Hierarchical non-linear bond graphs: a unified methodology for modeling complex physical systems," Simulation, 58 (4), pp. 230-248, 1992.

Cly90   Clymer J.R., Systems Analysis Using Simulation and Markov Models, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, Chapter 4, 1990.

Dor86   Dorf R.C., Modern Control Sytems, fourth edition, Addison-Wesley Publishing Co., Reading, Massachusetts, 1986.

Elz89   Elzas M.S., T.I. Ören and B.P. Zeigler (eds), Modelling and simulation methodology : knowledge systems' paradigms, North-Holland, Amsterdam, 1989.

Eth83   Etherington D.W., R. Reiter, "On Inheritance Hierarchies with Exceptions," Proceedings of the National Conference on Artificial Intelligence, Washington D.C., pp.104-108, 1983.

Fin79   Findler N.V. (ed.), Associative Networks, Academic Press, New York, New York, 1979.

Fis89a   Fishwick P.A., "Studying how Models Evolve: An Emphasis on Simulation Model Engineering," In Advances in AI and Simulation, R. Uttasingh and A.M. Wildberger (eds), Simulation Series 20(1), SCS International, San Diego, California, pp. 74-79, 1989.

Fis89b   Fishwick P.A., "Toward an Integrated Approach to Simulation Model Engineering," International Journal of General Systems, 17(1), pp. 254-287, 1989.

Fis91a   Fishwick P.A and R.B. Modjeski (eds), Knowledge-Based Simulation, Advances in Simulation 4, Springer-Verlag, New York, 1991.

Fis91b   Fishwick P.A. and P.A. Luker (eds), Qualitative Simulation Modelling and Analysis, Advances in Simulation 5, Springer-Verlag, New York, 1991.

Fis92   Fishwick P. A., and B.P. Zeigler, "A Multimodel Methodology for Qualitative Model Engineering," ACM Transactions on Modelling and Computer Simulation, 2(1), pp. 254-287, 1992.

For86   Forbus K.D., "Qualitative Process Theory," In Qualitative Reasoning about Physical Systems, D.G. Bobrow (ed), MIT Press, Cambridge, Massachusetts, pp. 85-168, 1986.

For90   Forbus K.D, and B. Falkenhainer, "Self-Explanatory Simulations: An Integration of Qualitative and Quantitative Knowledge," Proceedings of the National Conference on Artificial Intelligence, pp. 380-387, 1990.

Ghe91   Ghezzi C., M. Jazayeri, and D. Mandrioli, Fundamentals of Software Engineering, Prentice-Hall, Englewood Cliffs, NJ, 1991.

Gor90   Gorden R.F., E.A. MacNair, K.J Gorden, and J.F. Kuose, "Hierarchical Modeling in a Graphical Simulation System," Winter Simulation Conference, SCS International, San Diego, California, pp. 499-503.

Gra80   Graybeal, W.J., and U.W. Pooch, Simulation: Principles and Methods, Winthrop Publishers, Inc., Cambridge, Massachusetts, 1980.

Har89   Harrison W.H, J.J. Shilling, and P.F. Sweeney, "Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm," OOPSLA '89, Conference Proceedings, special issue of SIGPLAN Notices, 24(10), pp. , Oct, 1989.

Hay79   Hayes P.J., "The Logic of Frames," In Frame Conceptions and Text Understanding, D. Metzing (ed.), Walter de Gruyter and Co., Berlin, pp. 46-61, 1979.

Hop79   Hopcroft J.E., and Ullman J.D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Publishing Co., 1979.

Kal91   Kalasky D.R., and D.A. Davis, "Computer Animation with CINEMA," Winter Simulation Conference, SCS International, San Diego, California, pp. 122-127, 1991.

Kui86   Kuipers B., "Commonsense Reasoning about Causality: Deriving Behavior from Structure," In Qualitative Reasoning about Physical Systems, Bobrow D.G (ed), MIT Press, Cambridge Massachusetts, pp. 169-203, 1986.

Kui89   Kuipers B., "Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge," Automatica, 25(4), pp. 571-585, 1989.

Mil92   Miller V.T., and P.A. Fishwick, "Reasoning with Heterogenous Hierarchical Models," Applications in AI X, Orlando, Florida, April 1992.

Moo82   Moore R.C., "The Role of Logic in Knowledge Representation and Commonsense Reasoning," Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, Pennsylvania, pp. 428-433, 1982.

Myc84   Mycroft A., and R.A. O'Keefe, "A Polymorphic Type System for Prolog," Artificial Intelligence, 23(3), pp. 295-307, 1984.

Nan87   Nance R.E., "The Conical Methodology: A Framework for Simulation Model Development," In Methodology and Validation, O. Baki (ed), Simulation Series 19(1), pp. 38-43, 1987.

Öre89a   Ören T.I., "Bases for Advanced Simulation: Paradigms for the Future," Modelling and simulation methodology : knowledge systems' paradigms, North-Holland, Amsterdam, Chapter 1.2, 1989.

Öre89b   Ören T.I., "Simulation Models: Taxonomy," In: Encyclopedia of Systems and Control, M. Singh (ed.), Pergamon Press, New York, New York, 1989.

Öre91   Ören T.I., "Dynamic Templates and Semantic Rules for Simulation Advisors and Certifiers," Knowledge-Based Simulation, Advances in Simulation 4, Springer-Verlag, New York, Chapter 4, 1991.

Pag89   Page, T.W.,Jr., S.E. Berson, W. C. Cheng, and R.R. Monte, "An Object-Oriented Modelling Environment," OOPSLA '89, Conference Proceedings, special issue of SIGPLAN Notices, 24(10), October, 1989.

Peg90   Pegden C.D., R.E. Shannon, and R.P. Sadowski, Introduction to Simulation Using SIMAN, McGraw-Hill, Inc., New York, 1990.

Pet81   Peterson J.L., <u>Petri Net Theory and the Modelling of Systems</u>, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

Prä91a   Prähofer H., "System Theoretic Foundations for Combined Discrete-Continuous System Simulation," Doktor der technischen Wissenschaften, Dept. of Systems Theory and Information Engineering, Johannes Kepler University, Linz, Austria, 1991.

Prä91b   Prähofer H., "Systems Theoretic Formalisms for Combined Discrete Continuous System Simulation," International Journal of General Systems, 19(3), pp. 219-240, 1991.

Pui89   Puigjaner R. and D. Potier (eds), Modelling Techniques and Tools for Computer Performance Evaluation, Plenum Press, New York, 1989.

Ros89   Rosson M.B. and E. Gold, "Problem Solving Mapping in Object-Oriented Design," OOPSLA '89, Conference Proceedings, special issue of SIGPLAN Notices, 24(10), October, 1989.

Rot90   Rothenberg J., "Artificial Intelligence and Simulation (Tutorial)," Winter SImulation Conference, SCS International, San Diego, California, pp. 22-24, 1990.

Rum91   Rumbaugh J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson, <u>Object-Oriented Modeling and Design</u>, Prentice-Hall, Englewoods Cliffs, New Jersey, 1991.

Sch91   Sehriber T.J., <u>An Introduction to Simulation Using GPSS/H</u>, John Wiley & Sons, New York, 1991

Sev91   Sevinc S., "Theories of Discrete Event-Model Abstraction," Winter Simulation Conference, SCS International, San Diego, California, pp. 1115-1119, 1991.

Sta87   Standridge C.R, and A. B. Pritsker, <u>TESS: The extended Simulation Support System</u>, Halsted Press, New York, 1987.

Syd82   Sydow A., "Hierarchical Concepts in Modelling and Simulation," In Progress in Modelling and Simulation, F.E. Cellier (ed), Academic Press, London, Chapter 7, 1982.

Tho75   Thoma J.U., <u>Introduction to Bond Graphs and their Application</u>, Oxford, Pergamon Press, 1975.

Wym77   Wymore A. W., A Mathematical Theory of Systems Engineering: the elements, Rober E. Krieger Publishing Co., Huntington New York, 1977.

Wym86   Wymore A.W., <u>A Mathematical Theory of System Design</u>, SANDS, Tuscan, Arizona, 1986.

Zad75   Zadeh L.A., K. Fu, K. Tanaka, and M Shimura, Fuzzy Sets and their Application to Cognitive and Decision Processes, Academic Press, New York, 1975.

Zei76   Zeigler B.P., <u>Theory of Modelling and Simulation,</u> John Wiley, New York, 1976.

Zei84   Zeigler B.P., <u>Multifacetted Modelling and Discrete Event Simulation,</u> Academic Press, London, 1984.

Zei90   Zeigler B.P., <u>Object Oriented Simulation with Hierarchical, Modular Models,</u> Academic Press, New York, 1990 .
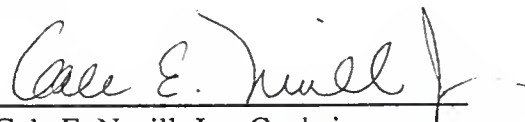
BIOGRAPHICAL SKETCH

Todd Miller received his Associate of Arts degree in architecture in 1982 from the University of Florida. In 1984 he received a Bachelor of Science degree in computer and information sciences through the College of Business Administration at the University of Florida. In 1988 he received a master's degree in computer engineering through the Computer and Information Sciences Department at the University of Florida. The completion of this dissertation will result in a long-awaited Ph.D. in computer science through the Engineering College at the University of Florida.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.
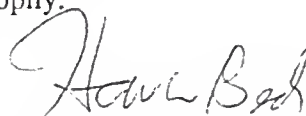
Paul A. Fishwick, Chairman
Associate Professor of
   Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Gale E. Nevill, Jr. , Cochair
Professor of Aerospace Engineering,
   Mechanics, and Engineering Science

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.
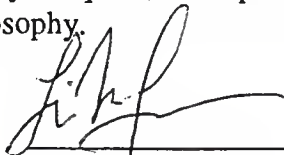
Howard W. Beck
Assistant Professor of
   Agricultural Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Richard E. Newman-Wolfe
Assistant Professor of
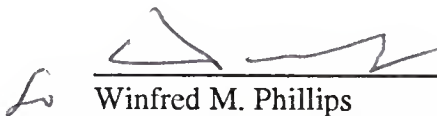   Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Li Min Fu
Assistant Professor of
Computer and Information Sciences

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

August 1993

Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School